DERIVATION

OF

LOGIC PROGRAMS

by

Christopher John Hogger

A Thesis Submitted

For The Degree Of

DOCTOR OF PHILOSOPHY

Of The

University of London

Imperial College of

Science & Technology

1978

ABSTRACT

Derivation of Logic Programs by *Christopher John Hogger*

The general theme of the thesis is the treatment of first order logic as a programming language. The subject is introduced by describing the way in which Robinson's resolution principle has enabled the construction of theorem proving interpreters which execute programs represented in clausal form. Kowalski's procedural interpretation of logic, which assigns operational significance to various properties of resolution refutations for logic programs, is then described in detail. There follows a whole chapter devoted to comparison of programming styles, discussing examples discovered by other researchers and also contributing some original ones.

After these preliminaries, the central subject of the thesis is introduced, namely the utility of the standard formulation of first order logic as a language for reasoning about the properties of logic programs. It is shown that clausal form is generally unsuitable for specifying, deriving, transforming and verifying logic programs, in contrast to standard logic which is eminently suitable for encoding the deductions which underlie these tasks.

It is then argued that the use of logic as a general computational language requires suitable inference systems for relating clausal-form programs to their properties expressed in standard logic. This leads to the formulation of a goal-directed quasi-computational inference system capable of deriving logic procedures from their specifications using just object level deductions; this is identified as a novel way of unifying the notions of synthesis and verification within a single technique.

It is shown that the inference system is adequate for deriving alternative representations of various well-known algorithms and is also capable of dealing with both procedures and data structures uniformly. The final chapter exploits the procedure derivation methodology to clarify logical taxonomic relationships within two algorithm families. Several sorting programs are derived from a single specification of sortedness, and several text-searching programs are likewise derived from basic properties of the substring relation. These derivations illustrate a number of interesting transformations which the inference system brings entirely within the scope of logical deduction in order to secure special kinds of algorithmic behaviour in the derived programs.

~~~~~~~~~~~~~~~~

To my Parents

~~~~~~~~~~~~~~~~

ACKNOWLEDGEMENT

I wish to record firstly an expression of appreciation to Dr. John Munro , *Reader in Civil Engineering Systems* in my Department, who invited me to apply for an academic appointment at Imperial College in 1971. It was his suggestion that I should undertake research for a doctorate and he has encouraged me to pursue its completion ever since. I am grateful also for the assistance which he has recently given me in order to present aspects of this research at two overseas conferences.

My early interest in computer programming methodology was much influenced by my contact with Professor M. M. Lehman in the Department of Computing and Control, who gave me much of his time during a period when my precise research intentions were still uncertain. In particular, his own interest in the problems facing the programming community in the construction of large-scale software implementation made me aware of some of the fundamental difficulties in the programming process to whose amelioration I hoped that my own studies might eventually contribute. During that time I was especially influenced by the views of other individuals associated with Professor Lehman's research project, and especially by Dr. A. L. Lim.

It was not until 1975 that I began to form a special interest in the development of new programming languages, when it was my privilege to meet Dr. Robert Kowalski shortly after he took up his appointment as *Reader in the Theory of Computing* in the Department of Computing and Control. Despite my limited background in Computer Science and complete inexperience in computational logic, he very kindly agreed to act as supervisor for the research which is now presented in this thesis. During the four years that have

passed since that time he has afforded me much advice and encouragement, and has patiently read and criticized a number of reports on my preliminary studies.

I have also benefited a good deal from contact with various members of the Theory of Computing Research Group set up by Dr. Kowalski soon after his appointment. Keith Clark, in particular, has influenced my views on several aspects of logic programming methodology, and I have learnt much from his contributions to that research group as well as to the published literature. Additional encouragement was given by Dr. John Darlington, whose fine work on program transformation has inspired many research efforts as well as my own; several of his published examples in this field have been adapted in this thesis to the logic programming formalism. My interest in programming style was particularly heightened by the work of Sten-Åke Tärnlund presented to the Logic Programming Workshop at Imperial College in 1976, and I was grateful for a recent opportunity to hold several discussions with him on the progress of logic as a programming language. Also at that time I enjoyed the kindness and hospitality of Wolfgang Bibel from the Institut fur Informatik in Munich whose work is quite close to our own.

Although my own studies have not been directly associated with any of the external research funding agencies, I should conclude here by acknowledging the support given by the Science Research Council to Robert Kowalski's project *Predicate Logic as Programming Language*; inasmuch as it has assisted that project, so it has also assisted those of us who have been fortunate enough to have had our researches enriched by Robert's vision and inspiration.

CONTENTS

INTRODUCTION

0.1 : BACKGROUND

Ever since the earliest developments of methods for programming computers through the use of symbolic input, computer scientists have been centrally concerned with the practical and theoretical attributes of the great number of programming languages which have been designed and implemented since that time. As the practice of professional computer programming has become ever more sophisticated and subject to increasingly stringent constraints upon the various parameters of program quality, so the demands made upon the designer of new languages have become more challenging. There are, of course, many differing views as to the ideal resources which a programming language ought to provide, even in respect of a particular problem domain. Yet it would seem unlikely that one could find serious dissent from a general goal of language design expressed more than thirty years ago (64) by, most fittingly, John von Neumann : that is, to provide the programmer with :-

> " *An effective and transparent logical terminology or symbolism for comprehending and expressing a particular problem, no matter how involved, in its entirety and in all its parts; and a simple and reliable step-by-step method to translate the problem (once it is logically reformulated and made explicit in all its details) into the code.*"

Computer programmers are still waiting for a transparent logical terminology like that advocated by von Neumann. In general, the languages which they employ scarcely differ, *in any fundamental sense,* from those used in the earliest days of computing. There are naturally many differences between their respective ways of describing a computational process intended to solve some problem of interest.

Yet their underlying philosophy is, for the most part, to describe a *process* rather than a *problem*. Consequently their semantics are specified in terms of the behaviour of abstract machines rather than in terms of what facts hold about the problem domain and about the particular problem at hand. These languages, then, share what might reasonably be regarded as an intrinsic incomprehensibility, in that they say very little that is explicitly meaningful about the problem, with the results that are now well documented within the annals of software practice : programs which are unclear, incorrect and resistant to confident modification.

The manner in which programmers actually do accomplish the composition of apparently correct programs is largely mysterious, even to themselves. We know that an Algol programmer can be asked to write a matrix multiplication routine and report back with the result within our lifetimes *even though he may afterwards admit that he has conducted no logical analysis of the program, during or after its composition, which associates its construction with the meaning of matrix multiplication*. His apparent success is due to substantial intuitive skills which enable him to bridge the two semantics which are associated, respectively, with the way his program describes a computational process and the way his specification describes a fact about matrices. If those skills were wholly reliable then we would hear much less controversy about language design and programmer education; in reality, of course, those skills are highly fallible because they are not well-founded.

It has been interesting to observe how, in the last decade, much greater emphasis has been placed upon the *logical* content of programs than hitherto. This can be observed especially in the proposals which have been made for the formal analysis of programs, where the objective is to show by sound deduction that a given program will compute some specified relation. It may also be seen in the development of informal methodologies such as structured programming which encourage programming styles intended to clarify the logical content of programs. These developments represent what we can interpret as successive approximations to von Neumann's ideal, namely the role of logic as the central system of reasoning in the programming process, rather than as just a peripheral tool.

The explicit manifestation of this role of logic has so far been mainly observed in its use as a specification language for conventional programming. Even in this respect it is scarcely used at all outside academic computing circles. This may be because the great majority of programmers are not logically literate, rather than because of the poor state of development of analytical devices such as program verifiers; if the importance of logic as a program reasoning tool was given greater emphasis in programmer education then more significant advances could be expected to follow in the provisions for assimilating its use into normal programming practice.

It is only comparatively recently that an even more interesting application of logic to programming has arisen, namely its use as a source language capable of automatic interpretation. This possibility has come about as a result of progress in mechanical theorem proving together with the creation of a remarkable procedural interpretation of first order logic developed by Robert Kowalski. During the last five years this interpretation has been used to establish a sound and convincing computational theory of 'logic programming', and has been implemented in a number of practical interpreters for logic programs. The most outstanding feature of logic as a programming language is its semantical independence of any execution mechanism conjoined with the fact that the source program statements which it affords comprise explicit assertions about the problem domain and the particular problem of interest; in other words, a logic program is meaningful in terms of the problem rather than in terms of the execution which will subsequently solve it.

It would be wrong to suggest that because the logic programmer can express the logical content of a computational problem explicitly that for him the question of correctness is inconsequential. Suppose, for example, that he required a program capable of deleting all occurrences of a member u from an input list. Then for computational purposes he might compose the following program statements :-

$$delete(u,\ u.nil,\ nil) \leftarrow$$

$$delete(u,\ v.x,\ v.x') \leftarrow u \neq v,\ delete(u,\ x,\ x')$$

Informally, these say that the result of deleting *u* from the unit list *(u)* leaves the empty list, whilst deleting *u* from a list which appends a list *x* to a unit list *(v)* such that $u \neq v$ is just the result of deleting *u* from *x* to leave *x'* and finally appending *x'* to *(v)*. We may ask whether this truly captures the notion which we associate with the symbol *delete*. Intuitively that notion requires that the members of the output list shall be exactly those of the input list other than *u* , and that they shall retain the same relative ordering in the output as they had in the input. Can it now be asserted with confidence that this coincides with, or is at least consistent with, the assertions made in the logic program ? Such questions are the concern of a fairly recently developed theory of logic program analysis, within which the concepts of termination, verification, synthesis and transformation are formulated upon a coherent logical foundation. The principal intention of the research reported in this thesis has been to contribute to this theory and to demonstrate its practical application.

0.2 : THE THESIS

Objectives of the Research

The purpose of the present work is to formulate and justify the concept of *logic procedure derivation* in support of the thesis :-

THESIS :

1. *The need for independent program specifications prevails in logic programming to the same extent as it does in conventional programming.*
2. *First order predicate logic provides an attractive specification language as well as a programming language.*
3. *First order deduction is sufficient for analysing relationships between programs and specifications.*
4. *Such analyses are practicable as well as theoretically well-founded.*

Logic procedure derivation refers to the task of showing that the statements (procedures) comprising the body of a logic program are true theorems about the problem domain implied by a first order axiomatic formulation of the problem which constitutes the program's specification. In practice this just amounts to constructing a series of deductions (a derivation) which treats the sentences in the specification as assumption formulas in order to prove each statement in the program. Because logic is a non-deterministic programming language, proof of each statement is logically independent of proofs of the other statements, and furthermore is independent of any assumptions about the behaviour of the program in execution; these circumstances confer a dramatic distinction between proofs of logic programs and proofs of conventional programs.

It is not only our purpose here to investigate the problem of verifying programs, important as this is. Logic procedure derivation can also be interpreted as synthesis (when the axioms used comprise just a naive specification) or as transformation (when the axioms comprise some other logic program's procedures, perhaps together with some other general facts about the problem domain). All these tasks are unified by their formulation in terms of proving computationally useful theorems implied by suitable axiom sets. Consequently it is reasonable to suppose that all may be accomplished through the agency of a single inference system for first order logic, and one of the intentions of the research reported here is to provide empirical evidence that this is indeed the case. Moreover, program transformation does not necessarily entail algorithm transformation; very often we may wish to modify the way in which a given logic program expresses the logic of some particular formulation of the problem at hand, perhaps with the object of exploiting an alternative control mechanism in the intended interpreter, or perhaps in order to obtain clearer logic. This may result in essentially the same algorithm (that is, run-time behaviour) but a substantial change in programming *style*. A variety of programming styles had been identified by other researchers before the present undertaking, and it is hoped that the latter will afford some clarification of the logic which underlies these kind of transformations.

Contribution of Original Material

There are two senses in which the author hopes that the thesis will afford evidence of an original contribution to the field of logic programming. Firstly, a great deal of work has been pursued by several other researchers on the methodology of logic programming since the inception of the discipline around 1973-74. Much of this remains unpublished or even undocumented in any comprehensive way, being disseminated amongst the various groups involved only through informal exchanges. It is true that there do exist a number of very useful reports explaining the computational theory of logic programming, amongst which are some fine publications by Kowalski, van Emden, Warren, Clark and Tarnlund; all of these are cited in the thesis and salient features from some of them are discussed here in detail. However, as far as the author is aware, no comprehensive and completed report describing the methodological advances in the last two years has yet been released. This is not to imply that the thesis captures the major part of those advances, but a considerable effort has nevertheless been made to do justice here in reviewing the contributions of others which have an especial bearing upon the central themes of the thesis. It should be mentioned also that new and substantial contributions to the literature of logic programming may soon be expected from Kowalski, who is preparing a book on the subject, and from Clark, who will doubtless be documenting his many researches in the field in his own forthcoming doctoral thesis *Predicate Logic as a Computational Formalism*. Jointly these should provide a fairly complete and up-to-date account of the work at Imperial College on the analysis of logic programs.

Secondly, all of the contents of Chapters 5, 6 and 7, together with parts of Chapters 3 and 4, are offered as the author's independent studies in logic programming methodology. The foundation for that material is established in Chapter 4 which explains the motivation and theoretical justification of logic procedure derivation. It must be declared that the concept of procedure derivation was also developed independently and contemporaneously by Clark, although our approaches to the technique have always differed. Clark's approach is very much aligned with that of Darlington's transformation system for sets of recursive function definitions in its emphasis upon that systems's

special rules for definiens substitution and its primary goal of securing recursions. The treatment of procedure derivation given here, by contrast, is more general in character and employs inference steps which are capable of a broader interpretation than those deployed in Clark's analyses. However, both of us were initially much influenced by Darlington's work, and it is additionally likely that the prospects of procedure derivation were anticipated long before by Kowalski. Chapters 5, 6 and 7 describe, respectively, the rules of inference identified by the author as having especial utility in the manipulation of standard logic in the course of deriving logic procedures; a collection of reasonably simple examples which may be viewed either as verifications or as syntheses; and two rather more concentrated studies of algorithm families intended to show the usefulness of the technique for program transformation.

0.3 : PREVIEW OF CONTENTS

Each chapter has been given its own preview in order to outline its essential contents, and so it is unnecessary to give a great deal of introductory detail here. Broadly the thesis can be viewed in four parts. The first of these is principally a survey of the general state of development in logic programming and spans Chapters 1, 2 and 3. Chapter 1 briefly surveys the contribution which theorem proving has made to computer programming, and explains how the theory of resolution proofs enabled logical deduction to be viewed as computation. The practical possibilities of resolution theorem proving for constructing conventional programs were examined in detail by Green, but it was Kowalski who formulated the procedural interpretation of logic which enabled resolution proofs to be treated as computations in their own right, thus establishing logic as a viable source programming language. Since resolution provides the basis of the current view of logic as a programming language, its relevant features are presented in Chapter 1. There, the syntax known as clausal form is introduced and used to illustrate the meanings of unification, resolution and refutation derivation which underly the operational meaning of logic programs. Finally a very

brief indication is given of the meanings of computation, algorithm, interpreter. output and non-determinism in terms of the proof-theoretic features of resolution.

Chapter 2 provides a more detailed description of the syntax, semantics and pragmatics of logic programs, together with some notational conventions adopted throughout the thesis. The procedural interpretation is presented and illustrated by an example. Here it is shown how mechanisms such as call activation, procedure invocation and data transmission can all be defined in terms of refutational theorem proving. The principal features of logic program interpreters are also introduced together with a detailed example which illustrates the significance of scheduling strategies. The chapter closes with a survey of some of the refinements which have been considered in order to improve upon the primitive default control mechanisms found in the Prolog-like interpreters commonly used at present.

Chapter 3 assumes that the essential theory of logic programming is understood, and proceeds to compare alternative styles in the composition of programs. The separation of logic from control is emphasized as the outstanding feature of the formalism, offering the programmer various ways of mixing those components in order to secure different algorithms or different representations of a given algorithm. Discussion is given of various kinds of procedure invocation, such as iterative and recursive mode, and different kinds of call activation, such as sequential and coroutined mode. It is shown how these kinds of behaviour may be procured through the agency of either explicit control mechanisms or special styles in the construction of the logic. Some rather exotic styles are demonstrated which enable top-down execution to emulate bottom-up execution, and an example is given of the application of this to the linear mathematical programming problem. There are many other special behavioural effects which can be induced through the correct choice of logic, and a few of these have been singled out for consideration here; many more will appear in later chapters. The important contribution of data structure choice to both programming style and computational efficiency is reviewed in the last section of Chapter 3. A wide selection of examples is given to show the effects of choosing different kinds of functional terms and sets of assertions, affecting, for instance, the question of whether

data can be retrieved by direct access or computed access; whether procedures can be invoked recursively or iteratively; and whether the accessing programs can be macroprocessed using appropriate data selector procedures.

The second part - Chapter 4 - deals with some of the techniques for reasoning about logic programs. Arguments are presented to justify the need for the standard formulation of predicate logic as a reasoning tool in addition to clausal form logic for mainly computational purposes. The early work by Clark and Tarnlund on termination and verification is given there together with some examples. A complete section is assigned to the discussion of the meaning of specification for logic programs, together with some conventions adopted for good specification style. The new technique of verification by procedure derivation is outlined in a section giving its theoretical justification and arguing its practical merits. The author's goal-oriented quasi-computational derivation style is presented there as the basis of the inference rules developed later on.

The third main part of the thesis is contained in Chapter 5. This explains the fundamental features and assumptions in the author's use of procedure derivation for analysing logic programs. Guidelines are given there for the composition and style of specification sets, together with some suitable conventions and terminology regarding the logistical aspects of the methodology. The two principal classes of inference rules are described in detail, explaining their differences and their cooperative interleaving during the derivation process. The most important rules for goal transformation - *modus tollens*, equivalence substitution and conditional equivalence substitution - are particularly emphasized. A complete section then surveys various ways in which the inference rules procure the derivation of typical recursive procedures and their bases, and the similarity of some of these applications to the Darlington transformation system is observed. The final section shows how the rules also apply to the derivation of low-level data accessing procedures, dealing firstly with access to terms and then with access to assertions; the latter discussion shows an interesting and instructive derivation which develops a list accessing procedure through several levels of abstraction.

The final part of the thesis comprises Chapters 6 and 7 which present examples of the application of procedure derivation. Chapter 6 examines six computational problems, developing various programs for them in the spirit of program synthesis. The first problem is the familiar one of list reversal, and derivations are given of both iterative and recursive programs. The second problem is that of searching a list for duplicates, and three algorithms of differing efficiencies are examined. The first effectively employs two independent iterative loops, the second makes the range of one of those loops dependent upon the progress of the other, and the third makes use of a stack to record the discovery of distinct members; all of these differing behaviours are secured by deriving appropriate procedures for a fixed control strategy. The next example deals with the generation of factorial tables, which may be computed either iteratively or recursively, with or without redundant multiplications and in either the natural order or the reverse order; again, all these behaviours are obtained satisfactorily using the derivation methodology. The problem of comparing the labelled frontiers of two binary trees is the subject of the fourth example, which exploits a simple associativity argument in order to secure the well-known but subtle algorithm which cooperatively transforms the trees in order to compare their first frontier labels. The fifth example is a simple addition problem over the elements of a matrix, but makes use of an interesting technique related to one of Kowalski's programming styles in order to develop an algorithm which computes a list of sums in quasi-parallel, in contrast to the naive but less efficient algorithm which computes the same sums sequentially. The final example is just the familiar eight queens problem. This was the first problem in logic programming ever studied by the author, and due to the attractive simplicity of its logic representation deserves a place in the thesis.

Chapter 7 is concerned with program transformation within two families of algorithms. The first section discusses the simplest sorting algorithm - 'naive-sort' - and derives its logic component from first principles. Then it is shown how additional information about the constructibility of lists enables an alternative derivation of 'merge-sort'. A series of transformations are shown which transform 'merge-sort' into 'quick-sort', 'merge-sort' into 'insert-sort' and finally 'quick-sort' into 'selection-sort'. All of the transformations use just the same inference rules as used for synthesis from basic

specifications. The final section considers rather more difficult algorithms intended for solving the text-searching problem. Several interesting alternative representations are given for the naive algorithm in increasing order of sophistication, until reaching one which can be transformed into either the linear Knuth-Morris-Pratt algorithm or the sub-linear Boyer-Moore algorithm. Whilst logical analyses of sorting algorithms have been developed by other researchers, the logical unification of the text-searching algorithms given here is, as far as the author can ascertain, a new contribution to the taxonomic analysis of that family.

The thesis is closed with a discussion of some related work by other individuals, not all of them using the logic programming formalism, and some views are given on the prospects of developing automatic tools for assisting derivations. Some suggestions are finally made concerning possible expectations for logic procedure derivation in the light of the experience described by the thesis.

CHAPTER 1

THEOREM PROVING AND COMPUTATION

PREVIEW

The central thesis of the logic programming formalism is that logical inference is amenable to a useful computational interpretation. That this concept can now be realized in terms of practical tools allowing the implementation of logic as a programming language is due to the successful results of research in automatic theorem proving. More specifically, the theory of logic programming is intimately associated with the theory of resolution proofs for first order logic. The first section of this chapter therefore begins with a brief account of the progress in automatic theorem proving which led up to the discovery of the resolution principle, and explains how this progress became relevant to the interests of computer programmers.

Amongst the early applications of resolution in connection with computer programming were implementations capable of synthesizing simple conventional programs from specifications expressed by axiom sets. In certain respects these might be viewed as the precursors from which present-day logic program interpreters evolved. However, the intelligibility of logic as an executable programming language came about not through advances in implementation technology but rather through the development of a convincing procedural interpretation of predicate logic. In order to properly appreciate the basis of this procedural interpretation it is firstly necessary to understand a limited part of the theory of resolution. The latter is briefly reviewed in the chapter's second section which introduces the notions of clausal form, unsatisfiability, unification and refutation derivation.

The final section then outlines the way in which various features

of resolution derivations can be interpreted computationally, thereby justifying the view of logic as a (non-deterministic) programming language capable of efficient implementation.

1.1 : HISTORICAL BACKGROUND

Early Objectives and Achievements

The study of automatic theorem proving during the last three decades reflects much earlier aspirations towards the systematization of mathematical proof. It is not surprising, then, that the earliest programmed proof procedures developed in the 1950's were applied most notably to mathematical theorem proving. This research was motivated by the hope that computers would provide proofs of significant theorems which would be too lengthy or too difficult to be undertaken by non-mechanical procedures. Computers could then be expected to accelerate the pace of mathematical discovery.

Apart from potentially contributing to the extension of mathematical knowledge, automatic theorem proving has also assumed importance in those aspects of the study of artificial intelligence which deal with the manipulation of knowledge by logical inference. There it has been successfully applied to such tasks as question-answering, game-playing and state-space problem solving. Theorem proving has proved useful in these various applications in consequence of the sufficient expressiveness of logic for representing knowledge and the efficacy of logical inference for processing it.

The first significant implementation of a theorem proving program was achieved by Newell, Shaw and Simon (65). This program was called the 'Logic Theorist' and was intended for generating proofs of formulas in the propositional logic. It was successfully used to prove various theorems selected from *Principia Mathematica* by goal-directed problem reduction. The Logic Theorist was later assimilated into the general problem solving system 'GPS' developed by Newell and his co-workers (66).

Propositional logic is too restrictive to serve as a convenient language for representing mathematical knowledge due to its lack of quantification. Most effort in automatic theorem proving has therefore been concentrated upon first order predicate logic (FOPL), which is adequate for representing all mathematics derivable from set theory. Some of the earliest algorithms for proving theorems in FOPL were proposed by Quine (69) and by McCarthy (58). McCarthy's paper outlined a proposal

for the construction of a theorem proving program called 'Advice Taker' whose fundamental inference system combined *modus ponens* with substitution of terms for variables. However, its intended capabilities could be enhanced by the user's provision of 'common sense' heuristics to guide the interrogation of an axiomatic data base describing the chosen problem domain. Implementation of this program was subsequently undertaken by Black (4) who incorporated its essential ideas intc a question-answering system. This system was able to solve some problems posed in propositional logic which had formerly defeated the 'Logic Theorist', but was nevertheless too inefficient to serve as a general purpose theorem prover.

By 1960 interest was growing in the search for uniform syntactical methods for proving theorems in FOPL, with the object of eliminating reliance upon semantic heuristics and other domain-specific devices designed for controlling theorem provers efficiently. Both Wang (81) and Gilmore (27) contributed programmed proof procedures for FOPL based solely upon syntactical rules. The behaviour of their programs, however, exhibited exponential dependence upon structural features of the input formulas representing the 'target' theorems, thus rendering the programs too inefficient for general application. A considerable improvement in performance was provided soon after by Davis and Putnam (21), whose program generated proofs with lengths only linearly dependent upon the number of variables in the input formulas. Nevertheless, each step in the proofs computed by it incurred a considerable computational burden, and Robinson (72) soon demonstrated some very simple formulas for which the Davis-Putnam program was quite infeasible.

It was not until 1965, when Robinson (73) published his discovery of the resolution principle, that efficient FOPL theorem provers appeared imminently feasible. Undoubtedly resolution provided a much stronger inference system than had been previously available; yet the problem of efficiently controlling the generation of proofs remained. There has subsequently been a great deal of investigation of heuristics for controlling resolution proofs, but not with sufficient success to fully realize the hopes of the mathematical theorem proving schools for efficient autonomous provers of 'hard' theorems. Despite this, resolution has contributed significantly to more specialized applications in computer science such as logic programming and the logical analysis and synthesis of conventional programs, wherein the necessary proofs are comparatively modest and (generally) foreseeable.

Theorem Proving and Conventional Programming

Computer scientists became especially interested in formalizing and proving properties of conventional programs after Floyd (26) showed how FOPL could be used to provide an axiomatic definition of their meaning. Although Floyd's proposals were focussed mainly upon the formalization of program semantics, they also provided an operational technique for proving programs to be correct with respect to axiomatic specifications. The progress of automatic theorem proving then became a matter of interest to the general programming community. King's thesis (42) describes a general purpose verifier for proving assertions describing flowchart programs. Program proving has since been investigated with great vigour and has an extensive literature; a good overview of the earlier work is given by London (55), and a more technical and up-to-date account by Katz and Manna (41). Despite the continuing interest in program proving amongst computer scientists, however, it would seem that programmers as a whole do not yet consider it a viable means of verifying their own 'real-world' programs. Generally they resort to testing methods instead. There are several factors contributing to this attitude, some of which are due to matters of programming psychology (described, for example, by Dijkstra (22)), whilst others may be due to insufficient appreciation of what can already be achieved with the verification tools now available. Underlying these factors is the fact that programmers do not normally view logic as the essential substance of their discipline, and so tend to be unconvinced of its usefulness. Nevertheless it would appear that computer-aided axiomatization and proof of conventional programs will not be capable of realistic assimilation into everyday programming practice until substantial improvements have been made in both programming languages and the styles in which their resources are deployed; these improvements will be necessary irrespective of the extent to which programmers are educated in the theory and pragmatics of logic.

Complementary to the task of proving that a given program conforms to some specification is the task of deriving the program from that specification. This process of program synthesis has also been studied with the aid of theorem provers. Green (30) has shown how a conventional assignment program can be constructed by examining the bindings of terms to variables in a resolution proof whose target theorem describes the program's intended input-output relation. The recovery of these bindings is the essence of the answer-extraction process which enables resolution to be used as a computational tool. Answer-extraction is dealt with in detail in Green's thesis (29) in connection with his work with Raphael

on the first implementation of a resolution theorem prover (28) as a question-answering system. A particularly good account of program construction by resolution is included in the book by Chang and Lee (11). Synthesis of conventional flowchart programs from resolution proofs was subsequently investigated by Lee and Waldinger (53). Their 'Prow' program-writing system suffered, like Green's system, the limitation of being unable to construct loop-containing programs. Methods for loop-construction were soon developed by Manna and Waldinger (60) by admitting induction axioms to the axiom set specifying the desired program.

The Origins of Logic Programming

Logic programming, which refers to the use of logic as a source programming language, has developed largely from progress in automatic theorem proving. In particular the current treatment of FOPL as a programming language derives from the computational features of resolution proofs. Whereas Green, Waldinger and others employed resolution proofs as the precursors for the construction of conventional programs, the logic programming formalism treats such proofs as computations in their own right. A sentence of logic may be looked upon as a source program intended for an interpreter consisting of a programmed proof procedure. Computation arises by the interpreter's construction of a proof of the input sentence, and the output of the computation is (generally) an accompanying set of bindings of terms to variables. Terms can therefore be regarded as the primitive data structures generated during computation, and the input sentences as procedures which process them. These notions are clearly closely connected with the answer-extraction process developed by Green. More general discussion of the relationships between logical inference and computation is to be found in papers by Hayes (33) and by Sandewall (76).

Kowalski's 1974 report 'Logic for Problem Solving' forms the earliest definitive account of logic as a programming language (49). Kowalski illustrates the richness of FOPL for representing problems in various ways and argues its merits as a machine-independent language suitable for the natural expression of deductive inferences made about computational problems. Computation is rigorously defined there in terms of resolution proof theory and then used to establish the important procedural interpretation of logic. A more concise summary is given in his paper to the 1974 IFIP Congress (50). A very satisfying account of the pragmatics of logic programming is also given by van Emden (23). At the present time (1978) these last two papers provide the most

comprehensive accounts of the foundations of logic programming to be found in the published literature.

The challenge of designing a practical interpreter in order to realize Kowalski's proposals was taken up by Colmerauer, Roussel and their colleagues, who successfully implemented a resolution interpreter for logic programs called 'Prolog' at the University of Aix-Marseille (17). Prolog, documented in greater detail by Roussel (74) and by Warren (83, 84), has exerted a strong influence upon subsequent implementations of logic interpreters in a number of schools of computing science and artificial intelligence.

In summary, then, the theoretical basis of the logic programming formalism owes much to earlier research (especially that of Robinson) in the application of automatic theorem proving to deductive problem solving, whilst its practical merits rest upon Kowalski's procedural interpretation and the efforts of those individuals who have given it expression in the construction of feasible interpreters.

1.2 : THE RESOLUTION PRINCIPLE

The general theory of resolution theorem proving would doubtless appear somewhat intimidating to the ordinary programmer whose notions of computation rest upon the simple machine-oriented actions underlying conventional programming language semantics. Fortunately, however, it is only necessary to become acquainted with the rudiments of resolution in order to understand how logic can be used for computation as well as for purely declarative purposes. Thus the following outline of resolution is restricted to deal with just those essential rudiments.

Validity and Undecidability

An important consequence of the treatment of logic program execution as a process of deductive theorem proving is that it necessarily confronts the central problem of any formal mathematical system, namely the problem of determining whether an arbitrary well-formed sentence in that system is a theorem, that is, provable. Godel's Completeness Theorem establishes that for FOPL this problem is equivalent to that of determining whether the sentence is valid, that is, true in all interpretations over all domains of interpretation.

The existence of an algorithm capable of totally deciding whether or not a sentence of FOPL is valid has been refuted by both Church and Turing. FOPL is therefore said to be undecidable. There exist subclasses of FOPL which are totally decidable, but these are too restrictive to be of practical value. There also exist partial decision procedures for FOPL which are able to decide the validity of a valid sentence, but which either fail to terminate or else terminate with no decision if presented with an invalid sentence. (Resolution, in fact, is a partial decision procedure for a particular subclass of FOPL.) FOPL is said to be semi-decidable by virtue of the existence of such partial decision procedures.

Unsatisfiability

The validity of a sentence can be investigated by considering the unsatisfiability of its negation; that is, whether its negation is false in all interpretations over all domains of interpretation. Clearly a sentence is valid if and only if its negation is unsatisfiable. Automatic theorem proving has most commonly been applied to the problem of investigating unsatisfiability as an indirect means of testing validity and in this guise is referred to as refutational theorem proving.

It is customary in logic programming to view a program as a set of sentences rather than as a single sentence. A logic program is then interpreted logically as the conjunction of its members. The set of sentences is described as unsatisfiable (or, equivalently, inconsistent) if and only if the conjunction of its members is unsatisfiable. The set is called satisfiable (consistent) if and only if it is not unsatisfiable (inconsistent). The equivalent meanings of satisfiability and consistency are just consequences of the Completeness Theorem which relates the model theory to the proof theory of FOPL.

Clausal Form Logic

The subclass of FOPL to which resolution is applicable is described as clausal form. The syntax of a sentence in clausal form is constructible from the following definitions :-

term : a constant symbol or a variable symbol or an n-ary function symbol followed by an n-tuple of terms;

atom : an n-ary predicate symbol followed by an n-tuple of terms;

positive literal : an atom;

negative literal : an atom preceded by the negation symbol;

clause : a disjunction of literals (possibly empty);

matrix : a conjunction of clauses;

universal prefix : a string of universal quantifiers;

clausal form sentence : a universal prefix followed by a matrix such that all variables in the matrix are quantified in the prefix.

Clausal form therefore describes those sentences in prenex-conjunctive normal form whose prefixes consist only of universal quantifiers. Systematic procedures exist for transforming any FOPL sentence to an equivalent sentence in clausal form; Nilsson's book (67) gives a clear account of one such procedure.

A sentence in clausal form clearly conjoins a set of clauses. Each clause is a sentence implicitly universally quantified over all the variables occurring in it. Treating a logic program as a set of clauses, the task of a logic program interpreter is to show that the set of clauses is inconsistent. The problem of showing that this is so is semi-decidable using an interpreter which implements the resolution principle.

Horn Clause Logic

A procedural interpretation of clausal form logic is especially simple to describe when it is applied just to a particular class of clauses known as Horn clauses. A Horn clause is defined as a clause which contains no more than one positive literal. Denoting a positive literal by L^+ and negative literals by L_1^-, ..., L_n^-, the Horn clause :-

$$L^+ \vee L_1^- \vee \ldots \vee L_n^-$$

is equivalent to the sentence :-

(i) $L^+ \leftarrow A_1 , \ldots , A_n$

where $\vee$ and $\leftarrow$ are the connectives 'or' and 'if', a comma is the connective 'and' and A_1, ... and A_n are the atomic parts of L_1^-, ... and L_n^-. Various special cases exist where there is no positive literal and where there are no negative literals; these are expressed in Kowalski's notation as follows :-

(ii) $\leftarrow A_1 , \ldots , A_n$

(iii) $L^+ \leftarrow$

(iv) $\square$.

Atoms appearing to the left and to the right of the connective $\leftarrow$ are respectively called the clause's consequent and antecedent atoms. The logical interpretations of these clauses are as follows, where X in each case denotes collectively the variables, if any, which they contain :

(i) for all X, $L^+ \leftarrow A_1 , \ldots , A_n$

(ii) for no X, $A_1 , \ldots , A_n$

(iii) for all X, L^+

(iv) false.

The restriction of logic programs to Horn clause form also simplifies the description of resolution (which is applicable to all clausal form sentences). Thus the following presentation of the resolution principle is conveniently restricted to Horn clause logic.

Unification and Resolvents

Unification is the process of determining a set θ of substitutions of terms for variables which, when applied to some given set of literals, yields a single substituted literal. For example, the substitution $\theta = \{ x:=c, z:=d \}$ is a unifier of the set of literals $\{ p(x,f(d)), p(c,f(z)) \}$ because its application to each literal yields the literal $p(c,f(d))$. If a set of literals has one or more unifiers, then there will exist amongst them a most general unifier. Informally, the most general unifier has the property that no other unifier for the set of literals is more simple. There exist algorithms which determine the most general unifier of any unifiable set of literals.

Unification of literals in Horn clauses forms the basic step in Horn clause resolution. Suppose θ is a most general unifier of the set $\{ L^+, A_k \}$ where L^+ is the consequent atom of one Horn clause and A_k is an antecedent atom of another. Then the resolvent of the two clauses is the unique clause obtained by substituting the entire antecedent of the first clause for the occurrence of A_k in the other, and applying θ to the result. The two given clauses are said to be resolved on literals L^+ and A_k. An example of the process just described is shown below.

Example :

first clause : $p(x,f(d)) \leftarrow q(d), r(f(x))$

second clause : $s(y,z) \leftarrow q(y), p(c,f(z))$

most general unifier : $\theta = \{ x:=c, z:=d \}$

resolvent : $s(y,d) \leftarrow q(y), q(d), r(f(c))$

The two given clauses are called parent clauses. It is important to observe that the resolvent is logically implied by the conjunction of the parent clauses, and that the resolvent of two Horn clauses is necessarily also a Horn clause.

The Empty Clause

The empty clause is generated as a resolvent in the special case where one parent Horn clause has no antecedent atom and the other has no consequent atom. An example is shown below.

Example :

first clause : $\leftarrow p(c,f(z))$

second clause : $p(x,f(d)) \leftarrow$

most general unifier : $\theta = \{ x:=c, z:=d \}$

resolvent : $\square$

Obtaining $\square$ as resolvent indicates that the parent clauses are inconsistent (contradictory). Using conventional exposition of the logic, it signifies that the sentence :-

$$\sim(\exists z)p(c,f(z)) \wedge (\forall x)p(x,f(d))$$

is false.

The Resolution Theorem

Given any set S of clauses, the resolution $R(S)$ of S is defined as the union of S with the set of all resolvents which can be obtained by resolving parents chosen from S. For any $n>1$, the set $R^n(S)$ is defined as $R(R^{n-1}(S))$. Robinson's Resolution Theorem establishes that S is inconsistent if and only if either $R(S)$ or some $R^n(S)$ contains the empty clause $\square$. The theorem therefore provides a single rule of inference (describing the generation of a resolvent) sufficiently powerful to demonstrate the inconsistency of S.

Resolution provides a more powerful inference system than those used by Gilmore, Davis and Putnam, whose methods relied upon successive

instantiations of the input sentence's variables by terms constructed from its functional vocabulary (the set of all constant symbols and function symbols occurring in the sentence). With these methods the input sentence could be proved inconsistent by discovering an inconsistent set of instantiations of it, by virtue of an important theorem due to Herbrand. Methods of this kind are called saturation procedures and are potentially combinatorially explosive, since the eligible set of terms (called the Herbrand universe) is generally infinite. The poor efficiency of saturation procedures is due to the lack of good criteria for choosing instances from the Herbrand universe. Resolution escapes these particular combinatorial difficulties by exploiting a more sophisticated rule for discovering falsifying instances for the input sentence.

Resolution Derivations

A resolution derivation from a set S of input clauses is a sequence of derived clauses $(C_1, \ldots, C_n)$ such that $C_1 \varepsilon S$ and every C_i $(i>1)$ is a resolvent of which each parent belongs either to S or to $\{ C_1, \ldots, C_{i-1} \}$. When this sequence has the additional property that every C_i $(i>1)$ also has C_{i-1} as a parent, it is called a linear derivation. If every C_i $(i>1)$ also has at least one parent in S then the sequence is called an input linear derivation: these are the derivations which are pursued by typical logic program interpreters.

Two kinds of linear derivation from Horn clauses are of especial interest and are described as top-down and bottom-up derivations. A top-down derivation consists solely of clauses having no consequent atoms; a bottom-up derivation consists solely of clauses having no antecedent atoms. The top-down/bottom-up distinction determines important differences in the ways in which resolution is used for problem solving.

The application of an inference system such as resolution to a set S of input clauses determines a space of all possible derivations from S. Within this space the derivations which terminate with the derived empty clause $\square$ (if any) are called refutation derivations.

Proof Procedures

A refutational proof procedure (which forms the core of any typical resolution interpreter for logic programs) augments a resolution inference system with a search strategy. The search strategy governs the way in which the proof procedure searches the space of derivations

determined by its inference system. The object of search is to find a refutation derivation, that is, to derive the empty clause □ from the input clauses and hence show that they are inconsistent. Efficient search strategies are an important requirement of practical interpreters intended for logic programs which admit more than one derivation.

Completeness and Correctness

An inference system is said to be complete if the space of derivations determined by any inconsistent set of input clauses contains a refutation derivation. It is said to be correct if it contains a refutation derivation only when the input clauses are inconsistent. In its most general form, resolution has been proved to be both complete and correct. However, when search heuristics are employed to guide resolution in ways which potentially restrict search to particular regions of the search space, completeness may not be preserved. Investigations of completeness and efficiency in a variety of proof procedures are reported in the doctoral theses of Kowalski (46) and Kuehner (52).

1.3 : COMPUTATION USING RESOLUTION

Computation and Algorithms

The computational theory of logic programming is based upon an operational interpretation of resolution derivations. A computation is represented by a linear derivation $(C_1, \ldots, C_n)$ in which every C_i $(i>1)$ has one parent chosen from the set of input clauses (the other parent being C_{i-1}). In particular, a refutation derivation ($C_n = \square$) represents a successfully terminating computation. If C_n cannot be resolved with any input clause then the derivation represents an unsuccessfully terminating computation. Moreover, a consequence of the undecidability of FOPL is that a resolution execution may not terminate at all.

A proof procedure associated with a particular set of input clauses constitutes an algorithm for generating computations from them. An implemented computer program which applies a search strategy with the resolution principle constitutes a general logic program interpreter. The logic programs which it interprets are just sets of input clauses.

Output from Computation

In general the objective in executing a logic program is not merely to confirm that the program is inconsistent, but also to discover instances of its variables which demonstrate the inconsistency. As each derived clause is generated during a computation, the unifying substitution which allowed its parents to be resolved may contribute to a set of bindings of terms to variables known as the binding environment of the computation. When computation terminates successfully the final state of this environment determines the desired instances of the variables of interest. A practical logic program interpreter will automatically output the final bindings of these variables if and when computation terminates.

Non-determinism of Logic Programs

A logic program exhibits non-determinism when its associated search space admits more than one derivation. The program determines neither the choice nor the order of derivations generated during computation. Instead these are determined by the search strategy employed to control the computation. The occurrence of more than one derivation in the search space is due fundamentally to the fact that FOPL describes relations rather than functions. In general, several input Horn clauses will be necessary in order to compute all possible members of any particular relation of interest, so that a derived clause C_i may resolve with more than one input parent and hence admit alternative choices for its successor C_{i+1}.

The efficiency of a general interpreter intended for processing non-deterministic as well as deterministic programs is strongly dependent upon its ability to apply intelligent criteria for choosing between alternative derivations. Furthermore, if the interpreter terminates a derivation unsuccessfully (that is, without deriving $\square$) then it should (ideally) be capable of applying an intelligent analysis of the cause of the failure in order to assist its choice of alternative derivations (if any) still awaiting exploration.

When the search space admits more than one possible refutation derivation, the possibility arises of alternative solutions to the problem described by the program. The choice and order of the solutions output from the computation is again determined by the interpreter's search strategy and not by the program itself. Other non-deterministic programs may have only one solution, yet allow this to be computed by significantly different refutations.

CHAPTER 2

FUNDAMENTAL FEATURES OF LOGIC PROGRAMS

PREVIEW

Chapter 2 presents the essential features of logic as a programming language. The first section explains the simple conventions adopted for representing programs and classifying their constituent parts. Of greater importance is the semantical description of logic programs given in the following section. Logic is unique as a programming language in that it has a model-theoretic semantics which makes no reference to any intended execution mechanism; this semantics endows a purely declarative meaning upon logic programs. By contrast, resolution theory provides the basis of an operational semantics which explains the meaning of logic programs in terms of what is computable (logically derivable) from them; this meaning is more akin to that normally assigned to computer programs. The procedural interpretation of logic is just one way of articulating such an operational semantics in terms of notions which prevail in other procedural programming languages. Moreover, because the procedural interpretation treats recursive Horn clauses as recursive procedure definitions, it is also possible to construct a fixpoint semantics for logic programs. The three kinds of semantics can be shown to be mutually equivalent in consequence of the Completeness Theorem for first order logic.

The third section describes the procedural interpretation in detail, introducing ideas such as program goal, program body, call activation, procedure invocation and data transmission. The effective control of these computational resources is closely bound up with the notion of scheduling, and a simple summation problem is examined which shows the

role of this in both top-down and bottom-up computations. In particular, last-in-first-out scheduling is the principal feature of the interpreters derived from the first significant implementation (Prolog).
Prolog is briefly described in a new section, and some Prolog-like computations are compared there for some problems concerned with addition over the integers. The final section considers some of the useful extensions to the elementary default Prolog strategy which have been proposed and implemented. The most notable of these extensions deal with coroutining, iterative invocation and intelligent backtracking.

2.1 : THE SYNTAX OF LOGIC PROGRAMS

Vocabularies for Syntax and Metasyntax

Throughout the thesis logic programs are assumed to be restricted to Horn clauses, the syntax of which has already been described in the previous chapter. Here it is only necessary to state the conventions which will be adopted herein for the vocabularies employed to construct Horn clauses and their metasyntax.

In the construction of Horn clauses :-

(a) commas and parentheses () are the only punctuation symbols;
(b) commas and ← are the only logical connective symbols;
(c) *i,j,k,u,v,w,x,y* and *z* (with arbitrary ornamentations) are the only variable symbols;
(d) all other lower-case alphabetic strings and all non-alphabetic strings may serve as function symbols or as predicate symbols.

In the metasyntactical description of Horn clauses :-

(a) as (a) above;
(b) as (b) above;
(c) *I,J,K,U,V,W,X,Y* and *Z* (with arbitrary ornamentations) are the only metasyntactic variable symbols;
(d) all other upper-case alphabetic strings and all non-alphabetic strings may serve as metasyntactic function symbols and predicate symbols.

These conventions are not intended to coincide with other presentations of logic programming, amongst which there is considerable variation.

The thesis also makes much use of non-clausal sentences in order to express facts about the problem domains investigated by the programs. Such sentences are presented herein using the orthodox notation for the standard formulation of first order logic. Thus in addition to the symbols for constructing Horn clauses, we shall also use the connectives and quantifiers :-

$$\sim \quad \vee \quad \leftrightarrow \qquad \forall \quad \exists$$

Note particularly that, throughout the thesis, the conjunction connective is represented by a comma. For both clausal and non-clausal sentences it will be permitted - where convenience dictates - to present both functions and predicates in infix notation instead of prefix notation.

Presentation of Logic Programs

A logic program is presented herein as a series of clauses. The presented ordering of the clauses has no syntactical significance, and no punctuation is employed to delimit individual clauses.

Syntactical Classification of Clauses

Horn clauses are classified according to their syntax as follows :-

(a) a clause with no consequent atom is called a *denial*;
(b) a clause with no antecedent atom is called an *assertion*;
(c) the clause with no atoms is called the *empty clause*;
(d) all other clauses are called *conditional assertions*.

2.2 : THE SEMANTICS OF LOGIC PROGRAMS

The Operational Semantics

The formal semantics of logic programs are developed in a paper by van Emden and Kowalski (24). They define the operational meaning of a logic program in terms of the members of relations (named by the program's predicate symbols) which are derivable from the program using some given inference system. This treatment is operational in the conventional sense in that the relations which the program computes are established by

reference to the computations (derivations) which it gives when executed by a specified interpreter (proof procedure). By interpreting derivations as computations, the operational semantics corresponds to the proof-theory of logic.

Kowalski's procedural interpretation of Horn clause logic, which forms the foundation of the logic programming formalism, treats atoms in a denial as procedure calls. The denial is a goal statement whose execution (through activation of the calls) computes instances for the variables occurring in it. Assertions and conditional assertions are interpreted as procedure definitions which may be invoked in response to calls activated from the goal. The procedural interpretation can be formalized easily in terms of resolution derivations and can therefore be regarded as one particular formulation of the operational meaning of logic programs. Because of its paramount importance in the computational interpretation of logic programs, a fuller discussion is deferred to a later section.

The Model-theoretic Semantics

Tarski's model theory of logic can be used to determine the meaning of a logic program in terms of the predicates which it logically implies. Because of the dependence of logical implication upon the notion of satisfiability (and hence upon the notion of a domain of interpretation), this treatment is essentially semantical, in contrast to the operational meaning of programs which would be traditionally viewed as belonging to the syntax of logic. However, in consequence of the completeness of FOPL, the operational and model-theoretic semantics are equivalent in the sense that they determine identical denotations for a given program's predicate symbols.

The Fixpoint Semantics

Van Emden and Kowalski also define a fixpoint semantics for Horn clause logic by interpreting sets of recursive conditional assertions as sets of recursive procedure definitions. By choosing monotonic transformations as mappings over Herbrand interpretations they establish equivalence between their fixpoint semantics and the model-theoretic semantics.

2.3 : THE PROCEDURAL INTERPRETATION

Program Goal

The goal of a logic program is, by convention, the unique clause in the program which has the syntax of a denial :-

$$\leftarrow G_1, \ldots, G_n$$

Denoting the goal's variables (if any) by $X_1, \ldots, X_m$ the goal represents :-

$$\sim(\exists X_1 \ldots \exists X_m)(G_1, \ldots, G_n)$$

and is treated as a refutable conjecture. The objective in executing the program is to discover instances of $X_1, \ldots$ and X_m which satisfy $(G_1, \ldots, G_n)$, that is, which provide a counter-example to refute the goal. When no variables occur in the goal, the intent of the program is simply to show that the goal is false.

Program Body

The set of all clauses in a program whose consequent atoms have the same particular predicate symbol R is called a procedure set for R. Each of its clauses is called a procedure for R. The body of the program is the set of all clauses in the program other than the goal, and is therefore just the union of all the program's procedure sets. The purpose of the program body is to assert knowledge about the problem domain investigated by the goal, and is assumed to be consistent. An example of a logic program is shown below.

Example :

```
← count(a.b.c.a.d.b.c.e.a.nil, w)

count(x,w) ← filter(x,y), kount(y,w)

filter(nil,nil) ←
filter(u.x',u.y') ← delete(u,u.x',z), filter(z,y')

kount(nil,0) ←
kount(u.y',w+1) ← kount(y',w)

delete(u,nil,nil) ←
delete(u,u.x',z) ← delete(u,x',z)
delete(u,v.x',v.z') ← u≠v, delete(u,x',z')
```

The program above contains four procedure sets associated respectively with the relations named as *count, kount, filter* and *delete*. The first

clause of the program is the goal, which conjectures that there exists no instance of *w* which is the *count* of the distinct members in the list *a.b.c.a.d.b.c.e.a.nil* . The computational properties of this program will be discussed in the next chapter.

Procedure Calls

In the procedural interpretation the antecedent atoms of a program's clauses are interpreted as procedure calls. A denial $\leftarrow G_1, \ldots, G_n$ is interpreted procedurally as a set of calls to the procedures named by the predicate symbols in G_1, ... and G_n. No logical significance is attached to the order in which the calls appear in a clause. The calls in the denial collectively constitute a goal whose solution requires the conjoint solution of the calls. Terms appearing in G_1, ... and G_n are interpreted as the arguments of the calls.

Procedure Definitions

The procedural interpretation assigns a computational meaning to factual assertions about the problem domain. Each clause in the body of a program is interpreted as a procedure definition. In a conditional assertion :-

$$A \leftarrow B_1, \ldots, B_m$$

the atoms B_1, ... and B_m are interpreted as a set of calls which constitutes the body of the procedure definition for procedure A. When there are no such atoms the body is empty. The atom A can be interpreted as a procedure heading which identifies the name of the procedure and its arguments. The order in which calls in its body (if any) appear has no logical significance. One way of reading the procedure is to say that the goal $\leftarrow A$ can be solved by solving the goal $\leftarrow B_1, \ldots, B_m$.

Activation of Procedure Calls

Activating a procedure call consists of selecting a call from the goal of the program with the object of initiating a computation which solves that call. Activating some call G_k in a goal $\leftarrow G_1, \ldots, G_n$ is therefore the process of initiating computation with the object of solving the subgoal $\leftarrow G_k$. In conventional programs this corresponds to the passing of control to a procedure call statement.

Activation of Procedure Definitions

Activating a procedure definition consists of selecting the procedure definition with the object of initiating a computation which derives a new procedure definition, that is, another fact about the problem domain. This has no analogue in conventional program execution.

Procedure Invocation

In conventional program execution, invocation means the passing of control to a procedure definition in response to a call activation. In logic program execution this process is emulated by top-down (goal-directed) invocation. However, it is also possible to invoke a logic procedure definition bottom-up by activating a procedure definition.

Top-down invocation of a procedure definition is the process of resolving it with an activated call in the goal by matching the call with the procedure heading through some unifying substitution θ. When a procedure definition $A \leftarrow B_1, \ldots, B_m$ is invoked in response to a call G_k activated from a goal $\leftarrow G_1, \ldots, G_n$, the resolvent is the new goal obtained by replacing G_k by the body $B_1, \ldots, B_m$ and applying θ to the result. Top-down invocation (corresponding to goal-directed problem solving) is the usual mode of invocation used in logic program execution.

Bottom-up invocation of a procedure definition is the process of resolving it with an activated procedure definition. When an invoked procedure $A \leftarrow B_1, \ldots, B_k, \ldots, B_m$ is resolved with an activated procedure $B \leftarrow C_1, \ldots, C_n$ by unifying the literals B_k and B with unifier θ, the resolvent is the definition obtained by substituting $C_1, \ldots, C_n$ for B_k in the first procedure and applying θ to the result. Bottom-up invocation derives a new fact from given facts and therefore corresponds to fact-directed problem solving.

The notions of activation and invocation described here must be carefully distinguished. Activation chooses a subgoal to be solved or a fact to be summoned; invocation chooses a procedure which responds to the activated call or fact.

Transmission of Data

The unifying substitutions which accompany procedure invocation can be interpreted as mechanisms for transmitting data between calls and procedure definitions. When a call G_k in a goal is matched with the

heading of a responding procedure definition, the variables in G_k (if any) are instantiated by terms supplied from the heading. Moreover, all occurrences of these variables in the goal are instantiated by these terms, so that data is distributed to other latent (unactivated) calls. The terms are interpreted as output data passed from the definition to the goal. Any variables occurring in the heading are simultaneously instantiated by terms supplied from the activated call. These terms are interpreted as input data passed from the call to the definition and thence distributed to all occurrences of those variables in the definition's calls.

An Example of Top-down and Bottom-up Computation

In the program below the predicate *sigma(z,w)* expresses the summation $w = (1+ \ldots +z)$ where z is assumed to be a natural number.

C1 : ← *sigma(3,w)*

C2 : *sigma(v+1,u+v+1)* ← *sigma(v,u)*

C3 : *sigma(1,1)* ←

Here the function symbol + is written in infix notation. For ease of presentation here, the symbols *2,3* ... etc. will be used to conveniently abbreviate terms like *1+1*, *1+1+1* ... etc. The goal of the program is to compute *w* as the sum of the first three natural numbers.

Top-down execution invokes *C2* in response to the activation of the call *sigma(3,w)* by resolving *C1* and *C2* to give :-

C4 : ← *sigma(2,u)*

The variable *v* in *C2* has been instantiated by the term *1+1* (=*2*) due to the input of the term *1+1+1* (=*3*) in the goal transmitted to the argument *v+1* of the heading in *C2*. The binding *v:=2* is distributed to all occurrences of *v* in *C2*, so that the partially instantiated term *u+3* is passed back from the heading's second argument as output to the goal variable *w*. Invoking *C2* a second time to solve the new goal *C4* similarly gives the resolvent *C5* :-

C5 : ← *sigma(1,u')*

with the binding *u:=u'+2* . Finally, invoking *C3* for the activation of the call in *C5* gives □ with the binding *u':=1* , so that the fully instantiated term *1+2+3* (=*6*) is computed for the goal variable *w*. The computation generated by this top-down execution is the refutation derivation *(C1,C4,C5,□)*.

Alternatively a bottom-up computation *(C3,C6,C7,□)* can be generated by resolving *C2* with successively derived assertions. Let *C3* be activated, thereby summoning a fact about the problem domain. Then *C2* can be invoked in response to this fact by matching its call *sigma(v,u)* to the heading of the activated procedure. The resolvent is a new fact about the problem domain :-

C6 : sigma(2,3) ←

Once again the procedure *C2* can be invoked, this time in response to an activation of *C6*. Resolving *C2* and *C6* in the same manner as previously with *C2* and *C3*, a further fact is derived :-

C7 : sigma(3,6) ←

Finally, *C7* is activated. However, suppose that the responding procedure *C1* is invoked instead of *C2*. *C7* and *C1* resolve to give the empty clause □ . This is just another fact about the problem domain - namely the fact that *C1*, *C2* and *C3* are inconsistent. Note that the final invocation transmits the term representing *6* to the goal variable *w*. Whereas the top-down execution computes *w* by successive approximations *w:=u+3* , *w:=u'+2+3* , *w:=1+2+3* , bottom-up execution defers instantiation of *w* until the final invocation.

Scheduling of Calls and Procedures

In the example above, efficient solution of the problem expressed by the input clauses required intelligent choices to be made between alternative responding procedures. For instance, the top-down execution chose *C2* in response to each of the first two goals *C1* and *C4*, but chose *C3* instead in response to the final goal *C5*. Likewise, the bottom-up execution chose *C2* in response to the first two facts *C3* and *C6*, but chose *C1* in response to the final fact *C7*. In neither regime were these the only possible choices : *C5* might have invoked *C2* instead, or *C7* might have invoked *C2*. In other problems a second kind of choice may also arise, namely the choice of which goal or fact to activate next. Both kinds of choice are clearly important to computational efficiency.

The usual way of dealing with alternative choices is to assign some schedule to them and then try each in turn. This is the way in which logic program executions normally proceed. When a goal contains several calls, the interpreter may assign to each a scheduling priority and thence determine the next call to be activated (that with the highest priority). If several procedure definitions respond to this call then the interpreter may assign to each a scheduling priority and thence

determine the next procedure to be invoked. With bottom-up execution scheduling priorities may likewise be assigned to determine the order in which facts are activated and the order in which responding procedures are invoked to deal with them. Scheduling priorities are governed by control information encoded within - or supplied by the user to - the interpreter.

Since the normal mode of execution is top-down, we shall normally assume just two kinds of scheduling to be of interest here; scheduling of calls in the current goal, and scheduling of the procedures which respond to them. Both kinds contribute to the inherent non-determinism of logic programs. Varying the activation or invocation schedules can influence either the efficiency of the ensuing computations or the output or both. Simple interpreters may process calls independently and in order of introduction to the goal, and may select responding procedures according to a fixed schedule assigned to the input clauses. More sophisticated interpreters can dynamically decide scheduling priorities during run-time and exploit properties of the current state of the computation in order to pursue this decision intelligently.

Because logic programs are non-deterministic, they leave open the choice of execution strategy. Whilst it is possible, in general, to choose scheduling strategies which emulate the kind of procedure invocation offered by conventional programming languages, logic programming admits more exotic possibilities which do not obtrude into the programming language itself. The richness of these possibilities is due largely to the great diversity in potential execution strategies provided by the interpreter , the choice of which is not constrained by the language's semantics. Indeed, procedure invocation from logic programs is more interesting than that from conventional programs even when it is limited to the simplest scheduling, since the successive approximation to output arising from the instantiation of latent calls means that activation of procedure calls and computation of output can be interleaved arbitrarily; a conventional procedure call does not usually return output until the computation which it instigates has successfully terminated, thereby deferring the activation of other latent calls.

2.4 : LOGIC INTERPRETERS

Prolog

Most implemented logic program interpreters are either direct versions of the Prolog interpreter written by Colmerauer and his colleagues at Aix-Marseille or else close derivatives from it. Prolog is essentially a top-down interpreter for Horn clause programs which, in its simplest mode of operation, employs a last-in-first-out strategy for selecting calls; that is, a strategy which always selects the most recently generated call in the goal as the next one to be activated. This schedule promotes a depth-first search through a subspace within the space of all top-down derivations determined by the program. The depth-first search necessitates provision for backtracking when derivations terminate unsuccessfully; Prolog is therefore often referred to as a 'top-down backtracker'. Implementation of both backtracking and recursive procedure invocation requires the interpreter to maintain a run-time stack, whose management critically determines execution efficiency. The stack records the history of call activations and procedure invocations which determines the current state of computation.

In addition to its default backtracking strategy, Prolog offers various devices to enable the programmer to specify further constraints upon the control of program execution; these permit, for example, run-time examination of bindings, discarding of branches in the search space and interpretation of unsuccessful derivations as proofs of negation. These control-determining devices are encoded within Prolog source programs as system-defined procedures (or 'evaluable predicates') in order to facilitate uniform parsing at compile-time. This is a rather unsatisfactory arrangement in that it is at variance with the central tenet of logic programming methodology, which is that matters of logic and control should be represented as distinct aspects of the programming process. Where it appears necessary for the programmer to provide explicit control information to the interpreter, this should be achieved without obtrusions into the logical text of the program. A more serious objection to some of Prolog's control directives is that they may alter the logical meaning of a program into which they are inserted, thereby potentially violating the first order semantics which underlies the logic programming formalism and guarantees its integrity.

Since its development at Aix-Marseille, Prolog has been implemented in various other institutions. At Edinburgh University Warren (84, 85) has established a version of Prolog which partially compiles logic programs for a DEC-10 computer; applications there include geometry problem solving (Welham, 87). Other versions have been implemented by M. Bruynooghe at the Katholieke Universitiet Leuve in Belgium and by B. Lichtman at Imperial College, London. P. Bonzon has written a Prolog-like interpreter in Pascal at the University of Lausanne in Switzerland. At Imperial College, a new Prolog-like top-down backtracking interpreter has been written in Pascal for the CDC complex there; this is described briefly in a report by Clark and Kowalski (14). This interpreter is free of the semantically doubtful features of Prolog and has a more presentable syntax for its input programs. It is also equipped with provisions for more sophisticated control mechanisms such as coroutining and intelligent stack management. Like Prolog, its simplest (default) mode of operation is sequential last-in-first-out scheduling of procedure calls, and responding procedures are selected in order of their presentation within the input program.

LIFO Scheduling of Procedure Calls

Last-in-first-out (LIFO) call scheduling plays an important role in logic programming by virtue of its simplicity as an execution mechanism. In assessing the practical merit of a program intended for an interpreter of the kind typically available at present, it is useful to consider firstly how well it behaves under the control of that interpreter's simplest strategy. If LIFO call scheduling produces inefficient behaviour from some initial ordering of the calls in the goal and procedures, the programmer may resort to a number of ways of seeking to improve this behaviour. In the first place it may be advantageous to merely change the ordering of the calls in the input program; this cannot affect the meaning of the program and is unlikely to change the intelligibility of its clauses very significantly, yet may produce dramatic changes in the ensuing computation. Alternatively the desired improvement may be achievable by re-ordering the presentation of the input clauses to the interpreter and thereby changing the scheduling of responding procedures. Both of these possibilities assume that the interpreter treats the ordering of calls and procedures as implicit control information, and both are attractive ways of improving behaviour because they preserve the meaning of the program. When neither is a sufficient remedy the programmer may be able to

influence computation through the agency of explicit control directives acceptable as input to the interpreter, and so override the default strategy. A further possibility, which is less attractive, is that the logic can be reformulated to give a more satisfactorily controllable description of the problem; sometimes this may require the use of logic which is less intelligible than the original version. Finally, and least satisfactorily, it may be necessary to enrich the interpreter's control strategy, a remedy which ought not to be within the province of the programmer; clearly this is more properly the responsibility of those concerned with implementation technology.

To illustrate the sensitivity of program behaviour to the ordering of calls, various goals are considered below for a simple program body in which the predicate *plus(x,y,z)* expresses *x+y=z* for integers *x,y* and *z*.

C1 : *plus(x,y,z)* ← *plus(x-1,y+1,z)*

C2 : *plus(0,z,z)* ←

C3 : *plus(x,y,z)* ← *plus(x+1,y-1,z)*

C4 : *plus(z,0,z)* ←

Clauses *C1 - C4* are sufficient to solve all solvable calls to the *plus* procedure, and therefore constitute what is referred to as a complete procedure set for *plus*. Now suppose that a solution is required to the equations :-

$$x + y = 6 \qquad x + 3 = 8$$

One way of expressing this problem is by using the goal :-

← *plus(x,y,6), plus(x,3,8)*

Prolog-like interpreters assume by default that calls are activated sequentially and independently from left to right as presented in the input clauses. Now if the call *plus(x,y,6)* is activated first and solved independently of the goal's second call, it is most likely to return an instance of *x* which will cause the second call to fail. In fact the first call might have to be re-activated by repeated backtracking many times under such a strategy, each time computing some new solution to the first equation, until discovering the successful instance *x:=5* . This would clearly be an extremely inefficient computation. Alternatively, submitting the re-ordered goal :-

← *plus(x,3,8), plus(x,y,6)*

would result in computation of *x* first, thus distributing the binding *x:=5* to the second call; then *plus(5,y,6)* would successfully return *y:=1*. This computation would require no backtracking.

Scheduling of Procedure Definitions

Scheduling of procedure definitions is as important to practical logic programming as the scheduling of calls. In the example just considered the clauses *C1 - C4* constitute a non-deterministic procedure set for *plus*. A given call *plus(x,y,z)* may compute various solutions depending upon which clauses are invoked in response to it. In fact *C1* and *C2* determine only triples *(x,y,z)* satisfying $x \geq 0$, $y \leq z$ whereas *C3* and *C4* determine only triples *(x,y,z)* satisfying $y \geq 0$, $x \leq z$. Thus if *C1* and *C2* are assigned higher priority than *C3* and *C4*, a call *plus(x,y,6)* will access solutions for *(x,y)* from the set { *(0,6), (1,5), (2,4), ...* } in preference to solutions from the set { *(6,0), (5,1), (4,2), ...* }.

Prolog-like interpreters assume by default that clauses are scheduled in order of their presentation within the input set. This ordering is normally fixed throughout execution, although Prolog does provide means of modifying procedures at run-time. Quite often it is difficult to specify a schedule which suffices for a wide range of possible invoking calls. For example, the call *plus(-1,2,z)* initiates indefinite recursion on *C1* if *C1* has higher priority than *C3*, whereas the call *plus(2,-1,z)* produces similar behaviour on *C3* if the priorities are reversed. Moreover, any call to *plus* generates indefinite recursion if both *C1* and *C3* (the recursion steps) have higher priorities than *C2* and *C4* (the recursion bases). Finally, if the invoking call is unsolvable such as *plus(1,2,4)* then execution will not terminate (even unsuccessfully) with any scheduling of *C1 - C4*.

If the intended interpreter is incapable of exercising any adequate scheduling of procedures to solve a goal satisfactorily (or indeed to abandon a goal satisfactorily) then the program has to be logically modified. The clauses below provide a more deterministic procedure set by introducing tests which 'block' fruitless computations.

C1' : $plus(x,y,z) \leftarrow x \geq 0,\ plus(x-1,y+1,z)$

C2 : $plus(0,z,z) \leftarrow$

C3' : $plus(x,y,z) \leftarrow y \geq 0,\ plus(x+1,y-1,z)$

C4 : $plus(z,0,z) \leftarrow$

Here the predicate $x \geq 0$ is an infix notation for a call to some procedure capable of determining whether *x* is positive. No call to *plus* can generate a non-terminating computation from the new procedures *C1' - C4*, irrespective of how the procedures are scheduled. This procedure set is suitable for calls in which the first two arguments are input integers. Other procedure sets for *plus* may be necessary to deal efficiently with invoking calls having different input-output arrangements.

Enhancements to Interpreters

Various enhancements have been proposed for improving the efficiency and intelligence of logic program interpreters. Although the simple control mechanisms afforded by the default Prolog strategy are adequate in many cases, there are other occasions when they prove inadequate to deal with the most favoured logical description of the problem. In such circumstances the programmer must either reformulate that description or else appeal to more sophisticated control mechanisms.

Inefficiency often derives from independent solution of problems which individually contribute to some common computational purpose. A typical example is the construction of a data structure X satisfying two properties P and Q. If this requirement is expressed using two calls $P(X)$ and $Q(X)$ then independent solution of them may result in repeated attempts by one call to construct an instance of X which also satisfies the other one. This is the kind of circumstance which may benefit from a coroutining facility. A coroutined execution of a set of calls is one which permits the computation instigated by each one of them to be intermittently suspended and resumed under the control of information received from the computations instigated by the others; this temporal interleaving of computations is typically regulated by the states of data structures to which the calls share access. In logic programs, calls which share variables may sometimes be profitably coroutined.

Kowalski (50) gives an example of coroutining for the goal :-

$$\leftarrow perm(S,y), ord(y)$$

which seeks an ordered list y which is also a permutation of the members of some set S. By activating the two calls independently in the given sequence, execution would repeatedly compute and discard complete but unordered permutations y of S until discovering one which happened to be ordered. A more efficient computation results when execution of the first call is suspended each time a new member of S has been selected to contribute to the construction of y, whereupon the second call is activated to determine whether addition of the new member to the partially constructed y preserves orderedness; if not, backtracking through one step in the computation from the first call enables selection of an alternative member. The coroutined execution is more efficient than sequential independent processing of the calls because the second call never receives an instance for y having more than one inversion of adjacent members; in effect, the second call behaves as a regulator controlling the output

of the first call. A study of the applicability of coroutining in logic interpreters is given in McCabe's MSc thesis (57). Apart from sequential and coroutined execution of calls, parallel processing is also possible when calls share no variables. When just one processor is available, such calls may be executed in quasi-parallel by suspending and resuming their executions arbitrarily, perhaps, for instance, to achieve useful space-saving economies in the management of the run-time stack.

Because logic possesses no means of specifying control information, the requirement that some procedure should be executed iteratively is not expressible within the text of a logic program. Instead, the programmer has to construct a recursive procedure and then require the interpreter to execute it iteratively. One way to do this is to execute the procedure bottom-up, as in the *sigma* example in the previous section. There, iterative summation of successive integers was accomplished by bottom-up invocation of a recursive *sigma* procedure. Alternatively one can resort to particular styles of recursive procedure which enable the interpreter to arrange for iterative implementation of the invocation mechanism whilst deploying top-down control. Examples of this are considered in greater detail in the next chapter dealing with logic programming style. In either case it is possible to mitigate the burden of stacking normally associated with recursive invocation by arranging that the interpreter reclaims the space allocated on the stack to an invoked recursive procedure when it discovers that each of its calls has been activated deterministically; the interpreter then knows that no record of that invocation need be maintained on the stack for the contingency of later backtracking.

In general it is advantageous to arrange that the interpreter maintains its binding environment in a data structure separate from that used to stack records of procedure invocations and their associated pointer systems. For example, Clark and Kowalski (14) show that a quasi-iterative execution of the recursive procedure :-

append(u.x,y,u.z) ← *append(x,y,z)*

to solve a goal like ← *append(a.b.nil,c.d.nil,z)* can generate a stack of bindings representing the incremental construction of the output argument *z* whilst overwriting another stack used for recording the history of procedure invocation. Moreover, they describe how in certain circumstances it is even possible to avoid stacking the bindings by using a data-overwriting mechanism which emulates conventional destructive assignment.

Clearly the provision for run-time stack management economies of the kind described above incurs a certain computational cost in monitoring the activation of calls to decide whether they are activated deterministically. With some logic programs it is possible in principle for the interpreter to conduct a compile-time analysis which shows that no stack is needed at all during the computation. For example, solution of the goal ← *append(a.b.nil,c.d.nil,a.b.c.d.nil)* with the procedures :-

append(nil,z,z) ←

append(u.x,y,u.z) ← *append(x,y,z)*

requires no stacking, and the interpreter should be capable of generating a truly iterative (rather than quasi-iterative) computation identical to that instigated by a compiled conventional program.

Efficient execution of non-deterministic logic programs requires intelligent control of the backtracking mechanism. The simplest backtracking interpreters preserve the stacked activation records generated by the computation induced by a call even after the call has been successfully solved, since the output which it distributes to other latent calls may cause them to subsequently fail; in that event sufficient information remains on the stack upon backtracking to enable the interpreter to decide which alternative ways remain for processing the earlier call. However, if the interpreter can recognise when a call can be processed deterministically then its activation record can be deleted as soon as it has been successfully executed. This can give substantial space reductions.

Sometimes it may also be possible to profitably rearrange the stack before backtracking after an unsuccessful computation. In particular it is advantageous to prevent backtracking from discarding useful results of computations which have not contributed to the cause of failure, and from pursuing alternative computations which necessarily fail for a common reason. The usefulness of analysing the cause of failure in order to decide the subsequent course of computation is examined in Hill's MSc thesis (34).

The question of whether or not a stacking mechanism is necessary depends upon the choice of problems to be solved and the choice of programming style. Many non-deterministic logic programs can be reformulated to give programs which are essentially deterministic, although the reformulations may be far from trivial and may result in substantially more complex descriptions of the problems of interest.

Most traditional programming is deterministic and iterative, and so it is possible to envisage restricted styles of logic programming which would meet most normal programming requirements and at the same time dispense with both search and recursion, thereby obviating the stack requirement. It is probably fair to say that the provisions of existing interpreters represent a view of computational problem solving rather more sophisticated than that actually encountered in current programming practice, thereby rendering those interpreters susceptible to difficulties in efficient implementation and hence uncompetitive against conventional program execution. On the other hand, successful remedies to those implementation difficulties would establish unrestricted Horn clause logic as a much more elegant and comprehensive problem solving language than those now in popular use.

The restriction of logic programs to Horn clause logic precludes the occurrence of negated calls. Yet sometimes it appears convenient to construct calls which investigate the non-membership of tuples in relations rather than their membership. For example, given a procedure *member(u,S)* for investigating members of a set *S*, one might wish to show that some instance α was not a member of *S*, by allowing a goal like :-

$$\leftarrow \sim member(\alpha,S)$$

In general the membership of a relation cannot be specified completely using just Horn clauses. However, Clark has investigated the semantical questions raised by interpreting failure of the goal :-

$$\leftarrow member(\alpha,S)$$

as a metatheoretic proof of $\sim member(\alpha,S)$ when it can be shown that all ways of investigating the call *member(α,S)* have been tried. Clark's investigation suggests that this metalogical inference may be a semantically acceptable extension to Horn clause interpreters, thus permitting them to process explicitly negated calls. The general problem of interpreting failure logically in order to solve negated subgoals is not special to logic programming; both Raphael (70) and Black (4) encountered the same issue in their deductive question-answering systems.

Most of the simple enhancements of Horn clause interpreters considered to date in connection with the Imperial College implementation are outlined in the report by Clark and Kowalski (14). It is likely that

their proposals will be implemented on the new interpreter there in the near future. At present it seems reasonable to expect that by intelligent control of non-determinism and efficient stack management future logic program interpreters will behave much better than Prolog. Warren's work at Edinburgh on the compilation of logic programs is also encouraging the view that it will not be too long before logic programs can be implemented as efficiently as conventional ones.

CHAPTER 3

LOGIC PROGRAMMING STYLE

PREVIEW

Good programming style is one of the defining attributes of a program's 'quality', and is as important in logic programming as it is in conventional programming. The art of good programming style is the art of finding a representation of an algorithm which allows its underlying concepts to be clearly perceived whilst ensuring that it remains computationally useful. A good program is therefore one which satisfies aesthetic as well as pragmatic criteria. In conventional programming these criteria are difficult to reconcile, as students of 'structured programming' will know only too well; here the central difficulty is in describing all the minutiae of the algorithm's control (to ensure efficiency) without committing the program's logical intentions to obscurity.

Possibly the most significant advantage of using logic as a programming language instead is that the meaning of the program is not dependent upon a specification of an execution mechanism. This permits the programmer to develop the program's logical and behavioural attributes in a more separable way than could be achieved for its counterpart written in a deterministic language of the kind now in popular use. The 'logic programmer' can experiment with significantly different Horn clause descriptions of the problem of interest and then consider, for each one, how to deploy the various control mechanisms of the interpreter at his disposal in order to secure an effective algorithm. Nevertheless it should not be falsely assumed that logic programs will therefore be *intrinsically* logically clear - it is not hard to find

published Prolog programs which are quite inscrutable at first sight. Logic offers substantial provisions for achieving logical clarity in programs, but does not *enforce* them.

Choice of programming style influences not only the effectiveness of algorithms and the clarity of program texts. It also influences activities such as the analysis and transformation of programs. The non-determinism of logic permits these activities, like that of composing programs, to deal with matters of logic and control separately. These arguments in favour of logic are presented in Kowalski's paper (51) describing the representation of algorithms as 2-tuples of the form *(logic, control)*. The merits of logic as a comprehensive computational tool which allows direct access to the *logic* components of algorithms are further expounded by Clark and Tarnlund (16). Investigation of a variety of computational problems by these researchers and others has contributed useful guidelines regarding the appropriate styles for logic components required for inducing particular kinds of behaviour in existing interpreters. Insofar as the primary purpose of the thesis is to consider ways of deriving useful logic programs, it is clear that awareness of the behavioural properties of various logic programming styles is important for discriminating between alternative program derivations.

The first section of the chapter concentrates upon the *logic + control* representation of algorithms. Examples of programs for solving a simple counting problem are firstly presented to show how the logic programmer can influence a program's behaviour purely through the agency of the logic component, and how by varying both logic and control he can obtain different representations of the same algorithm.

In the previous chapter it was mentioned that quasi-iterative invocation could be generated by employing appropriately styled recursive procedures. The present chapter considers this in greater detail by comparing two programs intended for list reversal. It is seen that transformation of recursive procedures to iteratively invokable form may require the construction of more elaborate procedures (in the sense of having more arguments and more subtle relationships between them).

Also of interest is the way in which top-down control applied to one procedure set can emulate the behaviour of another procedure set executed with bottom-up control. Such 'quasi-bottom-up' computations are exemplified by comparing alternative procedures for the highest-level description of the linear mathematical programming problem.

Unification is the fundamental device at the disposal of a logic program interpreter for the purpose of constructing and comparing data. Quite often it is possible to exploit unification in very subtle ways to provide algorithms which appear to require hardly any procedure invocation despite performing considerable computations. A few such algorithms are presented here which are interesting as slightly capricious novelties, although they cannot be advocated as examples of 'good' programming style.

The non-determinism of logic often encourages the use of special calls which do not significantly contribute to the logical description of the problem under consideration, but which nevertheless usefully constrain run-time behaviour. These are referred to here as 'control calls' and several examples are shown of their application. Likewise, it is sometimes convenient to constrain behaviour instead by the use of special procedure arguments (called 'control arguments'). Both of these methods for specifying control information implicitly through the agency of logic are just particular ways of exploiting the logic programming formalism's dependence upon pattern-directed procedure invocation.

The chapter's second section emphasizes relationships between procedures and data structure representations. It is shown how both terms and procedure definitions can be interpreted as data structures and how they dispose computation towards particular kinds of behaviour. Data structures may be directly accessible to some procedures but only indirectly accessible to others; the latter may only be able to access particular data components after a considerable amount of computation. The usefulness of indexing is mentioned for permitting direct access to procedure definitions. A number of interesting palindrome-testing programs are presented to illustrate the effects of choosing several alternative representations for the input data.

The section closes with a discussion of data abstraction through the use of selector procedures and the contribution which this technique makes to the clarity and flexibility of a program's high-level procedures.

3.1 : LOGIC AND CONTROL

Dependence of Behaviour upon Logic and Control

The variation in behaviour which results from varying the logic and control components of algorithms is indicated in the example below which deals with the problem of counting the number w of distinct members in some given input list x , for example *a.b.c.a.d.b.c.e.a.nil*. A conceptually simple algorithm is one which firstly filters out all duplicates from x to leave a list y and then computes the count w of members in y. The logic program below can be used to compute the relation *count(x,w)* in this way.

Program 1 : ← *count(a.b.c.a.d.b.c.e.a.nil,w)*

count(x,w) ← *filter(x,y)*, *kount(y,w)*

filter(nil,nil) ←
filter(u.x',u.y') ← *delete(u,u.x',z)*, *filter(z,y')*

kount(nil,0) ←
kount(u.y',w+1) ← *kount(y',w)*

delete(u,nil,nil) ←
delete(u,u.x',z) ← *delete(u,x',z')*
delete(u,v.x',v.z') ← *u≠v*, *delete(u,x',z')*

Here *filter(x,y)* holds when y is the list obtained by deleting all duplicates from list x (preserving the ordering of the remaining members); *kount(y,w)* holds when w is the number of members in list y; *delete(u,x,z)* holds when list z results from the deletion of all occurrences of member u from list x. When the program is executed with the default Prolog control, the ensuing algorithm suspends all counting until the task of filtering out all duplicates from the input list has been successfully completed. By delegating the tasks of counting and filtering respectively to two distinct procedure calls, a sequential LIFO interpreter like Prolog generates the computations for these tasks sequentially and independently.

By contrast, the program below dispenses with an explicit procedure for filtering out duplicates, and instead exploits knowledge

of the effect upon counting of deleting all occurrences of just one particular member. Given some input list *x* whose count *w* is desired, all occurrences of some member *u* are deleted from *x* to leave a list *z*; then *w* is computed as *1* plus the count of *z*.

Program 2 : ← *count(a.b.c.a.d.b.c.e.a.nil,w)*

count(nil,0) ←

count(u.x',w+1) ← *delete(u,u.x',z)*, *count(z,w)*

[together with the procedure set for *delete*]

A Prolog-like computation from this program has the effect of interleaving the tasks of deleting and counting members in the input list. Each recursive invocation of the *count* procedure contributes an increment of *1* to the cumulative evaluation of *w* and then deletes all occurrences of the member just counted. This program is more concise than the previous one, but is perhaps less obvious in its net effect.

The programs just discussed show how different algorithms may arise with a common control component (sequential scheduling) but different logic components (*Programs 1 & 2*). This indicates that the behaviour of algorithms should not be judged to be determined primarily by control information. Sequential scheduling is not the only control strategy worth considering for the counting problem above. The idea of interleaving counting with the deletion of duplicates suggests the possibility of solving this problem using coroutined procedure invocation. Suppose that a coroutining mechanism is used to control the activations of the two calls in the *count* procedure of *Program 1* :-

count(x,w) ← *filter(x,y)*, *kount(y,w)*

Activating the call *filter(x,y)* first, just one invocation of the recursive *filter* procedure is sufficient to partially construct the output *y* as some *u.y'* where *u* is the first member of *x* and *y'* is as yet undetermined. At this point control can be switched to the call *kount(y,w)* which counts the contribution of this *u* to *w* by recursing once on the recursive *kount* procedure. Counting the members of *y'* is then suspended until *y'* has been partially filtered by reactivating the first call. The task of specifying appropriate control information to the interpreter in order to achieve the effect of coroutining as described above is an easy one, since the suspension-resumption strategy here consists of no more than alternating from one recursive

procedure to the other, one invocation at a time. More generally, information for controlling coroutining has to be more elaborate than this simple arrangement in order to secure the computational economies which motivate its application; for example, the timing of suspension and resumption may depend in a complex way upon the instantaneous states of the data structures which the coroutined procedures jointly construct or interrogate.

It is interesting to observe that the algorithm having components *(Program 1, coroutining)* is identical to the algorithm having components *(Program 2, sequential scheduling)*. *Program 2* represents in logic the interleaving of two computations which *Program 1* induces sequentially in a Prolog-like interpreter. It is quite easy to explain the logical relationships between the two programs. Considering *Program 1*, suppose that *x* is the empty list *nil* in a call *count(x,w)*. The *filter* and *kount* bases can be resolved with the *count* procedure to give the assertion :-

$$count(nil,0) \leftarrow$$

which provides a procedure capable of directly counting *nil*. Now suppose instead that *x* is not empty, and resolve the *filter* and *kount* recursions with the *count* procedure to give :-

$$count(u.x',w'+1) \leftarrow delete(u,u.x',z),filter(z,y'),kount(y',w')$$

Now execution of *Program 1* by Prolog-like control will count the list *u.x'* by sequentially deleting *u* to leave *z*, filtering *z* to give *y'* and finally counting *y'* with the *kount* procedures. However, if the following additional knowledge is given :-

$$count(z,w') \leftrightarrow (\exists y')(filter(z,y'),\ kount(y',w'))$$

then it is clearly unnecessary to separately filter *z* and count *y'*, since solving *count(z,w')* will achieve exactly the same result. The procedure above with heading *count(u.x',w'+1)* can hence be written as :-

$$count(u.x',w'+1) \leftarrow delete(u,u.x',z),\ count(z,w')$$

which is exactly the procedure used by *Program 2*. In the absence of the assumed additional knowledge, no particular significance attaches to the process of invoking the *filter* procedure once and then invoking the *kount* procedure once; *Program 1* does not describe the contribution which such an invocation sequence makes to the progress of solving *count(x,w)*. *Program 2* exploits that knowledge by describing in its logic the fact that such a process contributes an increment of *1* to *w*.

Iterative and Recursive Procedure Invocation

Many of the commonplace relations dealt with by programmers are computable using iterative methods. In general, iteration is more efficiently implementable than recursion because completion of an iterative step - unlike a recursive one - does not depend upon the results of future steps. Iteration therefore avoids the stack management burden associated with recursive computation. Since Horn clause logic is a recursive programming language, it is important to arrange that interpreters execute recursive procedures in an iterative fashion whenever this is possible. Clark and Kowalski (14) have proposed an enhancement to the Imperial College interpreter which will implement invocation of a recursive procedure in a quasi-iterative manner when it can ascertain that all its calls in the previous invocation have been activated deterministically. To take advantage of this facility it is necessary to devise suitable procedure definitions for iteratively computable relations.

As an example, suppose that a program is required which, given some list *x* as input, computes the reverse list *y* as output. The program below describes this problem using the predicate *reverse(x,y)* to express the fact that *y* is the reverse of *x*, and the predicate *append*(z_1,z_2,*z*) to express that *z* is the list obtained by appending the list z_2 to list z_1; for the sake of example, *x* is chosen to be the list *a.b.c.d.nil* .

```
← reverse(a.b.c.d.nil,y)

  reverse(nil,nil) ←
  reverse(u.x,y) ← reverse(x,z), append(z,u.nil,y)

  append(nil,w,w) ←
  append(v.z',w,v.y') ← append(z',w,y')
```

When the recursive *reverse* procedure is invoked (assuming Prolog control) the only sensible choice of the first call to be activated is that to *reverse(x,z)* - calling *append(z,u.nil,y)* instead would result in a highly non-deterministic computation, because that call's input arguments do not constrain the choice of responding *append* procedure. Thus since the call to *append* cannot be activated deterministically before the next invocation of the recursive *reverse* procedure, the computation cannot be implemented iteratively; the program determines inherently recursive algorithms.

In contrast to the reversal program above, consider now the *reverse** program below which is capable of iterative execution :-

$$\leftarrow reverse^*(nil,a.b.c.d.nil,y)$$

$$reverse^*(y,nil,y) \leftarrow$$

$$reverse^*(x_1',u.x_2,y) \leftarrow reverse^*(u.x_1',x_2,y)$$

The predicate $reverse^*(x_1',x_2,y)$ holds when y is the reverse of the list obtained by appending x_2 to the reverse of x_1' ; to compute the list y as the reverse of list x , x is represented implicitly as the result of appending the goal's second argument to the reverse of the first argument. If the program is executed with Prolog-like control, that is, by top-down sequential LIFO scheduling, each invocation of the recursive *reverse** procedure generates a single new call to *reverse** ; when this call is activated it is necessarily done so deterministically, since no call to *reverse** in which the second argument is variable-free can invoke both the recursion and the basis. Consequently the stack's record for the next *reverse** invocation can overwrite the current record, so that no extension to the stack is required. Programs of a similar kind were demonstrated by Tarnlund (80) during a Logic Programming Workshop at Imperial College in 1976.

The *reverse** example illustrates a typical feature of transformations of recursive procedures to iterative form, which is the introduction of extra arguments to compute, in each iterative step, information which a recursive computation would encode as a stack of latent calls. Consider, for example, the recursive solution of the goal $\leftarrow reverse(a.b.c.d.nil,y)$. After two recursive invocations of the *reverse* procedure the goal becomes :-

$$\leftarrow reverse(c.d.nil,y''), append(y'',b.nil,y'), append(y',a.nil,y)$$

in which the two latent calls to *append* express the solution y as the result of appending $a.nil$ to the result y' of appending $b.nil$ to the reverse of $c.d.nil$; for brevity, denote this solution of y by the term $(y'':(b.nil)):(a.nil)$.

Now the associativity of the appending operation determines that the same solution y arises by appending $b.a.nil$ to the reverse of $c.d.nil$. This fact is exploited by the *reverse** procedure. After two invocations of this procedure the goal becomes :-

$$\leftarrow reverse^*(b.a.nil,c.d.nil,y)$$

In the first argument position the term $b.a.nil$ has already been constructed as a contribution to the partial evaluation of the

solution *(y":(b.nil)):(a.nil)* . A similar example is given by Clark (12) for the problem of computing factorials. He exploits the associativity of multiplication in order to derive a 3-place factorial procedure which partially evaluates the desired factorial in each iterative invocation rather than stacking multiplications in the customary recursive style.

Quasi-bottom-up Computation

Kowalski's paper (51) shows that the iterative behaviour which is typically obtained from conventional programs can be described in terms of bottom-up execution of recursive procedures. Whereas bottom-up procedure invocation in the Algol-like languages is precluded (due primarily to the irreversible nature of destructive assignment), logic procedures may be invoked in either top-down or bottom-up mode; their meanings are neutral with respect to the top-down/bottom-up distinction. At the present time there exist no logic interpreters capable of autonomously applying effective general strategies for controlling bottom-up invocation, although Prolog can be made to behave in a bottom-up fashion through the use of explicit control directives. For this reason it is interesting to discover that there exist logic programming styles which, with top-down control, mimic bottom-up computations; hence quasi-bottom-up behaviour can be obtained with existing interpreters through the agency of logic rather than control.

An example of a problem in which iterative behaviour is desirable is the general linear mathematical programming problem; here we consider how to use logic to represent the high-level procedures of the well-known *Simplex* algorithm. The objective of this algorithm is to derive a sequence of 'tableaux' $T_1, \ldots, T_n$ each describing the linear program's constraints and objective function, where T_1 is given and T_n satisfies an optimality criterion. Each computed tableau is derived from its predecessor using a matrix pivoting transformation, the mechanics of which are unimportant for the present discussion. It suffices here to assume that there exists some procedure *pivot(x',x)* which, given tableau *x'* as input, computes the successor tableau *x* as output. One possible formulation of the *Simplex* algorithm is then as follows :-

```
← tableau(x), optimal(x)

tableau(T1) ←
tableau(x) ← tableau(x'), pivot(x',x)

[together with procedures for optimal and pivot]
```

Here the metasyntactic symbol T_1 stands in place of some suitable term identifying or comprising the initial tableau. The call *optimal(x)* succeeds if an input tableau *x* satisfies the *Simplex* optimality criterion.

Prolog execution of this program is most unsatisfactory because of the loss of useful computation during backtracking each time some T_i fails to satisfy the optimality test. Suppose that a call *optimal*(T_i) has failed after the solution of calls *tableau*(T_{i-1}) and *pivot*(T_{i-1},T_i). The latter call will normally have instigated a considerable amount of computation which will be discarded upon backtracking, only to be recomputed during the subsequent computation of T_{i+1}. It is difficult to prescribe a simple general enhancement of the interpreter which would equip it to decide upon a sensible rearrangement of the stack prior to backtracking and hence avoid this loss; there appears to be no immediate means by which the interpreter could conclude that the computation from the call *pivot*(T_{i-1},T_i) is worth preserving.

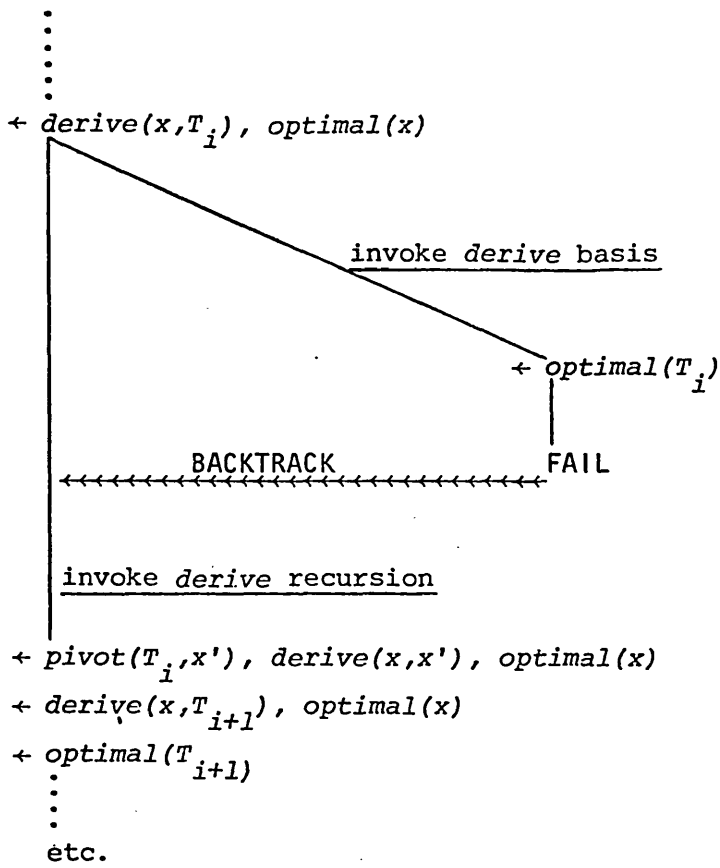
Much better behaviour can be obtained using the same control by introducing a new predicate *derive(x,x")* which holds if tableau *x* can be derived from tableau *x"* by a succession of calls to the *pivot* procedure. The goal of deriving optimal *x* from T_1 can now be pursued by the new problem formulation below.

```
← derive(x,T1), optimal(x)

derive(x,x) ←
derive(x,x") ← pivot(x",x'), derive(x,x')

[together with procedures for optimal and pivot]
```

Suppose now that some tableau T_i has already been computed, and that the goal is to derive an optimal tableau *x* from it. If T_i has not already been submitted to the optimality test then the first obvious possibility for *x* is just T_i. This possibility is explored using the *derive* basis procedure. Failing this, the other *derive* procedure can be invoked to compute T_{i+1} by pivoting T_i and then initiate a computation whose goal is to derive optimal *x* from T_{i+1}. This is illustrated by the following diagram of a region of the search space :-



It should be clear that in this computation the tableau T_i is not recomputed after failing the optimality test. The program behaves as an iterative generator of the sequence $(T_1, \ldots, T_n)$ with no redundant computation. To obtain similar behaviour from the previous program it would be necessary to invoke its recursive *tableau* procedure bottom-up, each time activating the most recently derived *tableau* assertion. This bottom-up computation would therefore generate a succession of assertions :-

tableau(T_1) ←
tableau(T_2) ←
⋮
etc.

in search of one which, when resolved with the goal, could transmit an optimal tableau to the call *optimal(x)*. A more detailed discussion of the logic representation of the *Simplex* algorithm and the use of the *derive* procedures to generate quasi-bottom-up behaviour is given in a report by Hogger (37).

Quasi-bottom-up behaviour has been independently investigated by Kowalski (51) and considered for the problem of path-finding in graphs.

A path from node *a* to node *b* can be explored in a variety of ways (depending upon the control strategy) using either the program :-

$\leftarrow go(b)$

$go(a) \leftarrow$

$go(y) \leftarrow arc(x,y),\ go(x)$

[together with *arc* assertions defining the particular graph of interest]

or else the program :-

$\leftarrow go^*(a,b)$

$go^*(x,x) \leftarrow$

$go^*(x,z) \leftarrow arc(x,y),\ go^*(y,z)$

[together with *arc* assertions defining the particular graph of interest]

Moreover, Kowalski also discovered a useful general relationship between these programming styles. An n-ary relation $R(X_1, \ldots, X_n)$ can be used to specify a $2n$-ary relation $R^*(X_1, \ldots, X_n, Y_1, \ldots, Y_n)$ as follows :-

$$R^*(X_1, \ldots, X_n, Y_1, \ldots, Y_n) \leftrightarrow (R(X_1, \ldots, X_n) \leftarrow R(Y_1, \ldots, Y_n))$$

Suitable instances of the definiens of R can then be substituted into this sentence in order to derive a recursive procedure for R^* whose top-down execution behaves like the bottom-up execution of a recursive procedure for R. Note that the above specification for R^* trivially implies the assertion :-

$$R^*(X_1, \ldots, X_n, X_1, \ldots, X_n) \leftarrow$$

which typically serves as the basis for the recursive R^* procedure. The general technique is also outlined by Clark and Kowalski (14) ; they make the interesting observation that the sentence relating R^* to R can be interpreted as the invariant of the loop associated with the conventional iterative program for R, which suggests a potentially useful link between logic program derivation and Dijkstra's calculus of invariants. Clark's paper (12) presents a derivation of the kind above, starting with the traditional 2-place factorial program and then deriving from it a 4-place factorial program whose top-down execution behaves like the former program executed bottom-up.

Exploiting Unification

The unification mechanism in a resolution interpreter can be regarded as a primitive processor for the class of data structures representable by terms. Superficially this processor is very limited in being capable only of lexically matching sets of unifiable terms. Nevertheless it is sometimes possible to exploit this capability in quite subtle ways, as will be shown presently. In particular, unification performs computational tasks which the programmer would otherwise (in the absence of unification) have to obtain through the use of explicitly programmed procedures. The unification mechanism can be looked upon as an implicit procedure which is automatically invoked to perform primitive data processing every time a user-defined procedure is invoked.

A trivial but instructive example is the problem of showing that two given lists are equal. This problem is expressible using the predicate *equal(x,y)* which holds when the lists *x* and *y* are equal. Suppose then that it is required to show that *x := a.b.c.nil* and *y := a.b.c.nil* are equal. One program for showing this is as follows :-

```
← equal(a.b.c.nil,a.b.c.nil)

  equal(nil,nil) ←
  equal(u.x,v.y) ← u=v, equal(x,y)
```

The algorithm obtained by employing Prolog-like control then arranges that the comparison of the input lists will be achieved by serially testing *a=a*, *b=b* and *c=c* ; these tests are conducted sequentially through the repeated calls to = (which typical interpreters can solve directly). Each invocation of the recursive *equal* procedure requires a very simple unification. The total work done by this algorithm has been explicitly discretized into a succession of elementary unification steps by the procedures written by the programmer. By contrast, consider now the following program for the same problem :-

```
← equal(a.b.c.nil,a.b.c.nil)

  equal(x,x) ←
```

Resolving the goal with the assertion instantiates both occurrences of *x* with the term *a.b.c.nil* ; the unification mechanism has to compare the two instances of that term in order to decide that both can be bound to *x*. Thus just one invocation instigates an algorithm for comparing lists using the interpreter's own built-in procedure for unification.

A number of ingenious ways of exploiting unification have been presented by Tarnlund (80). For instance, he has shown that for some purposes the customary recursive program for appending two lists (shown earlier in this section in connection with the list reversal problem) can be replaced by a single assertion :-

$$append^{\dagger}(w,v,w,v) \leftarrow$$

To append the list *d.e.f.g.nil* to the list *a.b.c.nil* just one invocation is sufficient in response to the goal :-

$$\leftarrow append^{\dagger}(a.b.c.x,d.e.f.g.nil,y,x)$$

which induces the bindings $v:=d.e.f.g.nil$, $x:=v$, $w:=a.b.c.v$ and $y:=w$; the required list *a.b.c.d.e.f.g.nil* is then output to the variable *y*. The single assertion is less general than the orthodox recursive *append* program in that it cannot be used to solve all input-output possibilities; for example, given an input list *w* it cannot compute as output two arbitrary lists w_1 and w_2 satisfying $append(w_1,w_2,w)$.

It is interesting to note that the $append^{\dagger}$ assertion is derivable in a manner similar to Kowalski's technique for obtaining quasi-bottom-up computations. For suppose that $append^{\dagger}(w,v,w',v')$ is specified in terms of the *append* relation as follows :-

$$append^{\dagger}(w,v,w',v') \leftrightarrow (\exists z)(append(z,v',w') \leftarrow append(z,v,w))$$

Then choosing $w:=w'$ and $v:=v'$ gives the desired assertion immediately. With reference to the problem above which is solved by this assertion, *z* is just the list *a.b.c.nil*. Thus Tarnlund's non-recursive $append^{\dagger}$ procedure can be regarded as just the basis part of a more general procedure set for $append^{\dagger}$ which exhibits quasi-bottom-up behaviour; this basis just happens to be suitable for appending two given lists by a clever arrangement of the invoking call's arguments.

Tarnlund has also found an interesting way of inserting an element into a list by exploiting subtle binding mechanisms. When the insertion position is known, it is convenient to specify the given list *x* implicitly in terms of two lists x_1 and x_2 satisfying $append(x_1,x_2,x)$ such that the inserted element *u* is to be inserted between x_1 and x_2. Then a rather orthodox rendering of the insertion problem is given by the following recursive program :-

$$\leftarrow insert(a.b.c.nil,d,e.f.g.nil,y)$$

$$insert(nil,u,y,u.y) \leftarrow$$

$$insert(v.x_1,u,x_2,v.y) \leftarrow insert(x_1,u,x_2,y)$$

Here the goal is to insert the element *d* between the third and fourth members of the list *a.b.c.e.f.g.nil*. This program repeatedly invokes the recursive *insert* procedure until all elements occurring in the first argument have been transferred to the fourth argument, at which point it is required to insert *d* between lists *nil* and *e.f.g.nil* ; the basis does this trivially, whence *y* is output as *a.b.c.d.e.f.g.nil*. Now suppose more generally that some list $u.x_2$ is appended to a list x_1 to give *y* ; then this can be regarded as insertion of *u* between x_1 and x_2. This intuition underlies Tarnlund's 5-place predicate *insert*(w,u,*x_2*,w,z)*, which expresses the fact that *w* is the result of appending *z* to some x_1 and also the result of inserting *u* between x_1 and x_2. Then the following equivalence holds almost trivially :-

$$insert^*(w,u,x_2,w,u.x_2) \leftrightarrow append^\dagger(w,u.x_2,w,u.x_2)$$

But this and the assertion $append^\dagger(w,v,w,v) \leftarrow$ jointly imply the assertion :-

$$insert^*(w,u,x_2,w,u.x_2) \leftarrow$$

Now this assertion is adequate for inserting *d* between *a.b.c.nil* and *e.f.g.nil* using the goal :-

$$\leftarrow insert^*(a.b.c.x,d,e.f.g.nil,y,x)$$

The goal resolves with the assertion to give *y:=a.b.c.d.e.f.g.nil* as output by virtue of the unifier θ = { *w:=a.b.c.x, u:=d,* x_2*:=e.f.g.nil, x:=u.*x_2*, y:=w* }.

These non-recursive programs are useful only when the desired input-output relations are computable by a single act of unification; their application is therefore very limited. Most commonplace relations computed by programmers seem to require repeated procedure invocation. For example, there seems to be no way of computing the reverse lists of arbitrary input lists except by using a recursive procedure to successively rearrange the members.

Control Calls

Control calls are procedure calls whose purpose is primarily to control execution in special ways, rather than to serve as an essential part of the logical description of the problem at hand. Such devices may be usefully employed, for example, to control backtracking and its effects upon the binding environment, or to control procedure invocation. Often their effect is to secure more deterministic behaviour than would be obtained in their absence. Control calls are different from control

directives such as are offered by Prolog in that they are *logical* constraints which indirectly influence control, having no special meaning from the interpreter's point of view.

An instructive example is the problem of searching a given finite set z for any two members u and v satisfying $u<v$. Let the predicate *pick(u,v,z)* express the fact that u and v are members of z such that $u<v$. Then a naive procedure for *pick* is as follows :-

$$pick(u,v,z) \leftarrow u \varepsilon z,\ v \varepsilon z,\ u<v$$

[together with procedures for ε and $<$]

Here the membership and comparison procedures are written in infix notation for clarity. Prolog-like execution of a program using the procedures above non-deterministically computes instances of u and v and then compares them. If the call $u<v$ fails then the interpreter backtracks to seek an alternative choice for v and reactivates the call $u<v$. If the choice of u happens to be the maximum member in z then this choice will also eventually have to be repealed after all instances of v have failed to satisfy $u<v$. Hence this computation may encounter a lot of backtracking.

Much better behaviour can be obtained by exploiting the knowledge that if any two members x and y satisfy $x<y$ then (x,y) is the desired solution, but if $x \nless y$ and $x \neq y$ then the solution is (y,x). Let the predicate *assign(x,y,u,v)* express the fact that the solution (u,v) is to be computed from (x,y) as just described. Then an alternative set of procedures adequate for investigating a call to *pick* is as follows :-

$$pick(u,v,z) \leftarrow x \varepsilon z,\ y \varepsilon z,\ x \neq y,\ assign(x,y,u,v)$$

$$assign(x,y,x,y) \leftarrow x<y$$

$$assign(x,y,y,x) \leftarrow y<x$$

[together with procedures for ε and $<$ and $\neq$]

Using these procedures, members x and y are selected non-deterministically and then passed to a test which checks that they are distinct; then they are passed to the call *assign(x,y,u,v)* ; the test $x \neq y$ ensures that the call to *assign* will succeed in computing a solution (u,v). Although the new arrangement is just as non-deterministic as the earlier one, the search space is now such that every possible selection of distinct members x and y results in a successful computation. The *assign* procedures logically encode useful knowledge about failure (in the sense that the instances causing the failure are used to infer a solution) and are therefore essentially concerned with control rather than logic.

Another interesting problem is that of showing that a given finite list x with distinct members consists of a list x_1 appended to a list x_2 such that x_1 is in strictly ascending order and x_2 is in strictly descending order; x_1 and x_2 may be empty lists as special cases. Introduce the predicate *updown(x)* to express this property of x; an example is *x:=1.3.4.8.6.5.2.0.nil* . A naive program for showing that this instance of x has the desired property is :-

```
← updown(1.3.4.8.6.5.2.0.nil)

updown(x) ← append(x1,x2,x), asc(x1), desc(x2)

asc(nil) ←
asc(u.nil) ←
asc(u.v.x) ← u<v, asc(v.x)        [and procedures for <]

desc(nil) ←
desc(u.nil) ←
desc(u.v.x) ← u>v, desc(v.x)      [and procedures for >]

append(nil,x,x) ←
append(v.x1,x2,v.x) ← append(x1,x2,x)
```

Now this program can give non-deterministic behaviour arising from the (generally) many ways of choosing x_1 and x_2 using the *append* procedures. If x has n members ($n \geq 0$) then there exist $n+1$ ways of choosing x_1 and x_2; but there exist no more than two of the choices satisfying *asc*(x_1) and *desc*(x_2) when x has the desired property. Prolog-like control is not a good strategy for this program because each time new choices of x_1 and x_2 are transmitted as output from the first call in the *updown* procedure, their orderedness has to be completely investigated by the calls to *asc* and *desc* even though many of their members will have been compared already in previous choices of x_1 and x_2.

When Prolog-like control is desired, the redundancies in the above program can be avoided by devising alternative procedures which interleave the decomposition of x with the task of comparing its members. The program below achieves this without introducing any new predicates.

```
← updown(1.3.4.8.6.5.2.0.nil)

  updown(nil) ←
  updown(u.nil) ←
  updown(u.v.x) ← u<v, updown(v.x)
  updown(u.v.x) ← u>v, desc(v.x)

 [together with procedures for desc, < and >]
```

Note that the *asc* procedures have now been dispensed with, their role being implicitly incorporated in the *updown* procedures. The role of the calls $u<v$ and $u>v$ is rather more subtle than in the previous *updown* program, where they served only to define the meanings of ascending and descending order. In the new program they continue to contribute to the logical description of the problem, but now also serve to control procedure invocation deterministically. Previously they in no way mitigated the program's non-determinism which came about through the use of *append* procedures to perform the decomposition of x; now this decomposition is put into effect by the last two *updown* procedures whose invocations are made mutually exclusive by the calls to < and >. The new program behaves excellently under Prolog-like control, successively inspecting pairs (u,v) satisfying $u<v$ and so confirming that x has a prefix $(1.3.4.8.)$ which is strictly ascending. When the eventual call *updown(8.6.5.2.0.nil)* is activated the call $8<6$ fails and so control is directed to the last procedure; this then instigates an iterative computation from the *desc* procedures to confirm that *6.5.2.0.nil* is strictly descending. Note that the program also fails efficiently if x does not possess the desired property; the previous program would inexorably try all possible decompositions of such an instance of x before terminating unsuccessfully.

Control Arguments

Procedure invocation can also be controlled by introducing special arguments for that purpose. Of course, procedure invocation in logic program execution is always pattern-directed, but here we are referring to arguments which are not components of the relations of interest but simply enforce various desirable attributes in the program's behaviour. This can be illustrated for the problem considered above. A new predicate is introduced whose second argument is a control argument in the sense intended here. Let the predicate *updown**$(x_2,\underline{asc})$ express the

fact that there exists an ascending list x_1 such that the list x obtained by appending x_2 to x_1 satisfies *updown(x)*. Likewise let the predicate *updown*(*x_2*,desc)* have an analogous meaning where x_1 is descending. Then the following program is sufficient for solving the problem :-

← *updown*(1.3.4.8.6.5.2.0.nil)*

updown(nil,z)* ←
updown(u.nil,z)* ←
updown(u.v.x,asc)* ← *u<v, updown*(v.x,asc)*
updown(u.v.x,asc)* ← *u>v, updown*(v.x,asc)*
updown(u.v.x,desc)* ← *u>v, updown*(v.x,desc)*

*[*together with procedures for < and >*]*

Every call to *updown** which is activated during computation has its second argument instantiated either by *asc* or *desc*. During discovery of the ascending prefix of *x*, this argument remains set as *asc*; as soon as the first descending pair *(8,6)* is found, the argument thereafter remains set as *desc* and hence confines invocation to the last *updown** procedure. The constant symbols act as control flags which effectively switch procedures 'on' and 'off' ; they exploit the fact that unification can only match identical terms. In the present example the control argument serves to divide the computation into two distinct and successive phases : the first phase is governed by the first two recursive *updown** procedures whilst the third recursive procedure is uninvokable ; the second phase is just the reverse of this, permanently switching 'off' the first two recursive procedures and generating computation just from the third one.

A more elaborate scheme for controlling procedure invocation with control arguments is shown in the next example, in which a control flag alternates between two states, so that each suspension of each procedure's eligibility for invocation is only temporary. Here the problem is that of splitting a given finite list *x* into two lists x_1 and x_2 consisting respectively of alternate members of *x*; for instance, if x = *1.2.3.4.5.nil* then x_1 and x_2 are respectively *1.3.5.nil* and *2.4.nil* . Let the predicate *transfer(x,*x_1*,*x_2*,y,z,w)* hold if lists *y* and *z* result from splitting some *x'* in the manner described whenever x_1 and x_2 result from similarly splitting the list obtained by appending *x* to *x'*; the splitting of *x* is such that its first

member is assigned to x_1 if $w=1$ but to x_2 if $w=2$. The following program effects the desired transfer of all members of x to x_1 or x_2 to produce the specified splitting of x :-

$\leftarrow$ *transfer*(1.2.3.4.5.*nil*,x_1,x_2,*nil*,*nil*,1)

transfer(*nil*,x_1,x_2,x_1,x_2,w) $\leftarrow$

transfer(u.x,x_1,x_2,y,z,1) $\leftarrow$ *append*(y,u.*nil*,y'),
transfer(x,x_1,x_2,y',z,2)

transfer(u.x,x_1,x_2,y,z,2) $\leftarrow$ *append*(z,u.*nil*,z'),
transfer(x,x_1,x_2,y,z',1)

[together with procedures for *append*]

The behaviour of this program with Prolog-like control can be seen quite easily from the refutation below; the list x' is shown next to each goal to clarify its role in the *transfer* specification above. For brevity, the calls to *append* are not shown but are assumed to have been processed.

$\leftarrow$ *transfer*(1.2.3.4.5.*nil*,x_1,x_2,*nil*,*nil*,1) x'=*nil*

$\leftarrow$ *transfer*(2.3.4.5.*nil*,x_1,x_2,1.*nil*,*nil*,2) x'=1.*nil*

$\leftarrow$ *transfer*(3.4.5.*nil*,x_1,x_2,1.*nil*,2.*nil*,1) x'=1.2.*nil*

$\leftarrow$ *transfer*(4.5.*nil*,x_1,x_2,1.3.*nil*,2.*nil*,2) x'=1.2.3.*nil*

$\leftarrow$ *transfer*(5.*nil*,x_1,x_2,1.3.*nil*,2.4.*nil*,1) x'=1.2.3.4.*nil*

$\leftarrow$ *transfer*(*nil*,x_1,x_2,1.3.5.*nil*,2.4.*nil*,2) x'=1.2.3.4.5.*nil*

$\square$ $\{x_1$:=1.3.5.*nil*, x_2:=2.4.*nil*$\}$

Here the last argument of *transfer* acts as a device for logically encoding control information for governing procedure invocation. It is easy to envisage other programs intended for Prolog-like interpreters that compute the new states of control flags by calling programmer-defined procedures to interrogate the current binding environment, thereby representing in logic the kind of decisions which a coroutining interpreter would implement through the control component.

Kowalski has employed constant symbols to improve programs like the one given earlier for solving *pick*(u,v,z). He observes that when the call $x<y$ fails for some pair of members x and y selected from z, backtracking to an alternative procedure poses the problem of showing $y<x$. In general, it may be computationally expensive to attempt

solution of both components of an *if-then-else* construct. He argues that it may be better to compute a solution *true* or *false* encoded by a control argument and then exploit unification to control procedure invocation. Doing this for the *pick(u,v,z)* problem would lead to the procedures :-

pick(u,v,z) ← *x*ε*z*, *y*ε*z*, *x*≠*y*, *less(x,y,w)*, *assign*(x,y,u,v,w)*

assign(x,y,x,y,true)* ←

assign(x,y,y,x,false)* ←

[together with procedures for ε, ≠ and *less*]

Here it is assumed that procedures for *less* will efficiently compute *w:=true* if *x<y* and *w:=false* otherwise. The use of the call to *less* dispenses with the need to interrogate the < relation twice as in the earlier procedure set which used the *assign* procedures.

3.2 : DATA STRUCTURES

Terms and Procedure Definitions as Data Structures

The primitive data structures manipulated by logic programs are terms. These can be used to represent entities such as sets, lists, trees and arrays which are traditionally dealt with by programmers. Terms which contain no variables may be regarded as wholly determined data structures. However, terms which do contain variables may also be transmitted and manipulated by procedures in ways having no direct analogy in conventional programming languages.

Semantical descriptions of conventional languages customarily distinguish between data structures and procedure definitions, treating them as distinct kinds of computational resource. The meanings of procedures are explained in terms of their competence to interrogate and generate data structures, but they are not also expected to interrogate or generate other procedures. However, all sentences in logic programs can be interpreted either in the normal way as procedure definitions or else as data structures in their own right. Moreover, execution of logic programs (notably using bottom-up invocation) can result in the run-time generation of new sentences representing either new data structures or new procedure definitions. Kowalski's paper (51)

alludes to the terminological confusion which could arise in attempting to apply to logic programs those views of algorithm structure which treat procedures and data structures as fundamentally distinct.

A simple example which illustrates the flexibility of logic for representing data is one which deals with lists. For simplicity the example is restricted to lists in which no member has more than one occurrence. Suppose that a procedure set is required which investigates the relation of consecutivity between members of some given list. The relation of interest can be expressed using the predicate *consec(u,v,z)* which holds when *v* is consecutive to *u* in the list *z* (that is, *v* is the immediate successor of *u* in *z*). If the lists which the desired procedure set investigates are to be constructed from the orthodox constructors . and *nil* then the following sentences comprise a complete procedure set for *consec* :-

consec(u,v,u.v.z) ←
consec(u,v,w.z) ← *u≠w*, *consec(u,v,z)*

These provide a description of consecutivity which is applicable to all lists represented by the chosen class of terms. To investigate a particular list *L = (a,b,c,d)* , *L* is represented by the term *a.b.c.d.nil* and transmitted as input to the procedures. Querying the consecutivity of members *c* and *d* in *L*, for example, would then require repeated procedure invocation to solve the appropriate goal ← *consec(c,d,a.b.c.d.nil)*.

An alternative way of representing a list makes use of a set of sentences which assert the list's consecutive members. For instance, *L = (a,b,c,d)* is representable by the set of three assertions :-

consec(a,b,L) ←
consec(b,c,L) ←
consec(c,d,L) ←

These specify the list *L* uniquely subject to the assumption made earlier that no member has more than one occurrence in *L*. Investigating consecutivity in *L* now consists of a search amongst these assertions, treating them as individual components of a data structure. The assertions explicitly express the logical consequences of the general *consec* procedures above applied to the specific list *L* represented by the term *a.b.c.d.nil*; the assertions are derivable from them by bottom-up invocation.

Yet another way of investigating consecutivity is by comparing the positions of members in the list. Let the predicate *item(u,i,z)* express the fact that *u* is the i^{th} member of list *z*. Then to show that some *v* is consecutive to some *u* in *z* we can invoke the procedure :-

consec(u,v,z) ← *item(u,i,z)*, *item(v,i+1,z)*

If the list *z* is to be represented by orthodox terms then its members and their positions are computable using the following procedure set for *item* :-

item(u,1,u.z) ←
item(u,i+1,v.z) ← *item(u,i,z)*

These, together with the *consec* procedure which they serve, allow the positions of given *u* and *v* to be computed and compared to test for consecutivity. Alternatively a particular list *L = (a,b,c,d)* can be represented by the set of *item* assertions :-

item(a,1,L) ←
item(b,2,L) ←
item(c,3,L) ←
item(d,4,L) ←

which are derivable by bottom-up invocation using the general *item* procedures above applied to the specific list *L = a.b.c.d.nil* .

The use of assertional data structures, that is, data structures represented by sets of assertions or conditional assertions, in logic programming is discussed by Kowalski (49), who gives an elegant example of their application to grammatical analysis of sentences represented by chains in labelled graphs. More recently (51) he has shown that conditional assertions can be usefully employed in path-finding algorithms, representing each arc by a sentence of the form

$node(n_2)$ ← $node(n_1)$

in place of the more obvious representation :-

$arc(n_1,n_2)$ ←

A variety of interesting algorithms can be obtained by combining these representations and their associated accessing procedures with different kinds of control strategy.

Data Access

Choice of data structure representation naturally influences the design of procedures intended for processing them and is therefore an important aspect of programming style. When terms are used as data structures, programming style is disposed towards computations which recursively assemble or disassemble the terms; such computations clearly require efficient management of procedure invocation and binding environments to be of practical value.

Consider the problem of accessing a list representation using Prolog-like control in order to discover which member (if any) is consecutive to a given member. One algorithm for this task is that which firstly locates the given member, infers the position of its successor, if any, and then looks up that successor. Suppose we try to do this using the general *item* procedures given earlier, choosing the specific list *L = (a,b,c,d)* represented by a term and the specific given member *c*. In this event the program is as follows :-

```
← consec(c,v,a.b.c.d.nil)

consec(u,v,z) ← item(u,i,z), item(v,i+1,z)

item(u,1,u.z) ←
item(u,i+1,v.z) ← item(u,i,z)
```

The resulting computation is quite inefficient because the two calls to *item* activated from the *consec* procedure are almost identical. The call *item(c,i,a.b.c.d.nil)* searches *L* (by recursive decomposition) to discover *c*'s position *i:=3* ; the next call *item(v,4,a.b.c.d.nil)* searches L (again by recursive decomposition) to look up its 4^{th} member. Clearly there is much computational redundancy in the two searches through *L*'s members.

A much better way of solving the goal above is to use instead the general *consec* procedures for the term representation :-

```
consec(u,v,u.v.z) ←
consec(u,v,w.z) ← u≠w, consec(u,v,z)
```

The ensuing computation maintains a pair *(c,v)* in the binding environment until such time as the basis can be invoked in response to the call *consec(c,v,c.d.nil)* ; now that *c* matches the first member of the last argument (implicitly computing its position), the basis provides direct access to *c*'s successor *d*.

Yet another way of solving the above problem is to represent *L* by a set of *consec* assertions as shown previously; then no other procedures are necessary to discover *c*'s successor. With this arrangement the interpreter never has to manipulate terms representing the list fragments *(b,c,d)* and *(c,d)* as is the case with the two preceding programs. In the simplest accessing regime for the present program the interpreter will just conduct an iterative search through the *consec* assertions seeking one which immediately solves the goal; in this event the binding environment remains vacuous until the solution is found.

Of course, the use of assertional data structures does not imply that access must involve search. A powerful enhancement to elementary interpreters is the facility for accessing individual assertions directly by exploiting special arrangements in the binding of data to physical memory. Indexing and hash-addressing are obvious potential techniques for this purpose. Indexing is already employed extensively amongst computational systems which rely upon pattern-directed invocation of data or procedures, and is especially useful when the latter can be arranged in some practical and natural ordering. For the current problem the *item* assertions representing *L* can be conveniently ordered using their second argument position as the key position; then to solve the call *item(v,4,L)* the interpreter could directly access the 4th assertion and so discover the member *d*. More intricate accessing mechanisms are necessary for dealing efficiently with other input-output permutations of the argument positions. Sophisticated accessing protocols are clearly essential to the manipulation of large-scale collections of data such as are found in data base query systems, but are also essential to quite routine computational tasks. For instance, the inversion of a modestly-sized matrix by a logic program would necessitate the use of an assertional representation of the matrix emulating the traditional array (a term representation being wholly unviable), and efficient access would be a crucial feature of the inverting algorithm. Much useful material on logic and data bases can be found in the papers presented at a Workshop on Logic and Data Bases held at Toulouse in 1977 (90).

Terms are unsuitable as run-time representations of data structures when their inherent syntactical properties obstruct convenient access to the components of interest, since such obstruction generates expensive computational penalties. Consider now a new problem which is that of

showing two given finite sets to be equivalent, this being expressed by the predicate *equiv(x,y)*. The simplest term representation of sets uses two constructors, say $\emptyset$ and : , where $\emptyset$ represents the empty set and a term *u:x* represents the set $\{u\} \cup x$. With this arrangement the set {*a,b,c,d*} has 24 distinct representations. Suppose that some computation activates a call *equiv(x,y)* with *x* instantiated by the term *a:b:c:d:*$\emptyset$ and *y* by *d:c:b:a:*$\emptyset$. To show that these terms represent equivalent sets it is necessary to search them for common members, which is expensive since the members are not directly accessible. To consider this in a little more detail, suppose that the problem is solved using the procedure :-

equiv(x,y) ← *subset(x,y)*, *subset(y,x)*

where *subset(x,y)* expresses $x \subseteq y$. The *subset* calls can in turn be investigated using the procedures :-

subset($\emptyset$,*y*) ←
subset(u:x,y) ← *u*ε*y*, *subset(x,y)*

*u*ε*u:y* ←
*u*ε*v:y* ← *u*≠*v*, *u*ε*y*

These recursive procedures are typical of those needed for accessing the constituents of terms. The call *equiv(a:b:c:d:*$\emptyset$*,d:c:b:a:*$\emptyset$*)* eventually instigates a call *a*ε*d:c:b:a:*$\emptyset$ which is only solvable by recursing on the ε procedure; similar calls are made to investigate both the membership of *b*, *c* and *d* in the set represented by *d:c:b:a:*$\emptyset$ and the membership of *d*, *c*, *b* and *a* in the set represented by *a:b:c:d:*$\emptyset$. These calls to ε obviously incur substantial computational costs.

A somewhat better way of solving the set equivalence problem is by employing the procedures below :-

equiv($\emptyset$,$\emptyset$) ←
equiv(x,y) ← *union*(u,x',x)*, *union*(u,y',y)*, *equiv(x',y')*

where *union*(u,x',x)* holds when *x* is the set $\{u\} \cup x'$. Suitable procedures for *union** are as follows :-

union(u,x',u:x')* ←
union(u,v:x',v:x)* ← *u*≠*v*, *union*(u,x',x)*

Applying Prolog-like control to these procedures gives a better algorithm than with the procedures which interrogate the *subset* relation because as soon as some member *u* is selected from *x*, a search is made for *u* in *y*; when this succeeds, the computation effectively deletes *u* from *y*

in order to dispense with the need to check subsequently that *u* is a member of *x* in the course of showing that all members of *y* belong to *x*. The previous algorithm is inefficient in that it does not exploit its solution of *subset(x,y)* such as to avoid redundant membership tests when solving the call *subset(y,x)*. Nevertheless both approaches suffer the cost of indirect access to the sets' members.

In the equivalent sets example the choice of terms to represent sets introduces unwanted structural properties into the data structure representations. The problem of showing two sets to be equivalent does not logically require the notion of an inherent ordering of their members. Yet the terms constructed using $\emptyset$ and : inherently order their constituents, so that programs which access them have to confront this ordering even though it has no logical significance for the problem at hand. Structurally the terms are more complicated than the data structures which they represent.

In general one might expect that list-like terms are especially suitable for representing lists. This is certainly true of many problem formulations, but not so of others. A variety of examples for palindrome-testing are now considered which illustrate several ways of combining list representations with accessing procedures. The examples will demonstrate procedures which provide for computed access to terms, then a use of terms which places the burden of access upon the interpreter's unification procedure, then the benefits of choosing a term representation different from the orthodox one and finally programs which access assertional list representations using both computed and direct access.

Informally a list is a palindrome if its first and last members are identical and when a palindrome remains after deleting those two members; this remaining list is called here the 'middle' of the original list. Also the empty list and all unit lists are defined to be palindromes. These stipulations can be summarized more precisely by the sentences :-

palin(x) ← *empty-list(x)*
palin(x) ← *unit-list(x)*
palin(x) ← *first(x,u)*, *last(x,u)*, *middle(x,x')*,*palin(x')*

Now these sentences can be regarded as procedures for solving calls to *palin*, provided that procedures are also devised for dealing with the selector calls to *empty-list*, *unit-list*, *first*, *last* and *middle*.

The procedure set for *palin* above is logically neutral to the choice of representation for lists, but its computational usefulness depends upon how efficiently the selector procedures can be implemented for whatever representation is eventually chosen. Consider firstly the position if orthodox terms are used, that is, terms constructed using . and *nil*. This representation allows trivial procedures for some selectors :-

empty-list(nil) ←
unit-list(u.nil) ←
first(u.x,u) ←

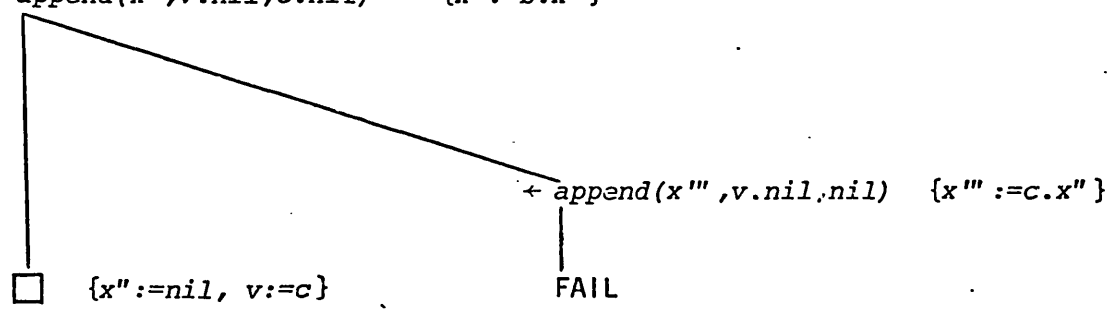
but precludes direct access to the 'middle' and 'last' components. The latter can only be obtained by computed access. One fairly concise way of computing them is by using the familiar *append* procedures as follows :-

middle(x,x') ← *append(u.x',v.nil,x)*
last(x,u) ← *append(z,u.nil,x)*

*[*and the usual procedures for *append]*

To access the middle or last components will then clearly require repeated procedure invocation to disassemble the term passed to ***x***. Inspection of the procedure set for *append* will show that the last member of *x* can be computed iteratively, and that most of the invocations needed to extract the middle of *x* can also be implemented iteratively. In fact the computations involved here are wholly deterministic except for the very last *append* invocation used in computing the middle of ***x***; this is illustrated by the following computation which seeks the middle of the list *a.b.c.nil* :-

← *middle(a.b.c.nil, x')*
← *append(u.x',v.nil,a.b.c.nil)*
← *append(x',v.nil,b.c.nil)* {*u:=a*}
← *append(x",v.nil,c.nil)* {*x':=b.x"*}



The branch appears in this search space because the call at its root can invoke either the *append* basis or the recursion.

Despite the possibilities for iterative procedure invocation afforded by this way of solving *palin(x)*, the algorithm as a whole is clearly too inefficient to be useful. Accessing the last and middle components of *x* is not only indirect but, worse still, engenders much duplication of effort; for instance, investigating *x = (a,b,c,d,c,b,a)* will require construction of the unit list *(d)* in the course of computing each of the middles of *(a,b,c,d,c,b,a)* , *(b,c,d,c,b)* and *(c,d,c)*.

A rather better algorithm may be obtained using the same data structure representation but quite different procedures. The logical specifications for the relations *palin* and *reverse* can be shown to imply the procedure :-

palin(x) ← reverse(x,x)

where *reverse(x,y)* holds when *x* and *y* are mutually reversed lists. Now it has already been shown that the standard procedures for list reversal :

reverse(nil,nil) ←
reverse(u.x,y) ← reverse(x,z), append(z,u.nil,y)

must give rise to recursive invocation if the second argument in the invoking *reverse* call is an output variable. Here, however, both calling arguments are input lists, and in this circumstance it is easy to show that computation will be mostly deterministic if the call to *append* is always activated before the call to *reverse(x,z)*. A wholly deterministic computation can alternatively be secured by using the *reverse** procedures instead which were discussed in the previous section. Then the question of whether or not *x* is a palindrome can be answered by the procedures :-

palin(x) ← reverse(nil,x,x)*
reverse(y,nil,y) ←*
reverse(w,u.z',y) ← reverse*(u.w,z',y)*

As execution proceeds with this program, the reverse list of *x* is gradually constructed in the first argument position of the calls to *reverse** in such a way that each new member contributing to this construction is obtained from the *beginning* of the second argument and affixed to the *beginning* of the first one; thus there is never any need to access the last member of any term, which was the main source of inefficiency in both the *last* and *middle* procedures considered earlier. Altogether this formulation of the palindrome problem is satisfactory both aesthetically and pragmatically for the data structure representation.

It is interesting to find that there exists an even simpler program for palindrome-testing which is also deterministic and very efficient. Its simplicity in appearance is due to its reliance upon a single unification to perform the necessary comparison of the input list's 'left' half with its 'right' half. The rest of the computation is only concerned with assembling the two halves of *x*. The intuition behind the algorithm is that any palindrome *x* must be constructible by finding some list *z*, reversing it and then appending to the result either the same list *z* or the list *u.z* where *u* is an arbitrary element. For example, if *x = a.b.c.c.b.a.nil* then *z* is the list *c.b.a.nil* ; if *x = a.b.c.b.a.nil* then *z* is the list *b.a.nil* and *u* is the element *c*. This particular decomposition of *x* can be expressed by the predicate *palin*(z',z)* which holds when the result of appending *z'* to the reverse of *z* is a palindrome *x*. The problem can then be solved using the procedures below :-

palin(x) ← *palin*(x,nil)*

palin(z,z)* ←
palin(u.z,z)* ←
palin(u.z',z)* ← *palin*(z',u.z)*

For instance, the computation required to show that *a.b.c.b.a.nil* is a palindrome proceeds as follows :-

← *palin(a.b.c.b.a.nil)*
← *palin*(a.b.c.b.a.nil,nil)*
← *palin*(b.c.b.a.nil,a.nil)*
← *palin*(c.b.a.nil,b.a.nil)*

decomposing *x* into two lists

□ comparing the two lists.

This algorithm can obviously be implemented iteratively. Moreover, it terminates by comparing *b.a.nil* with *b.a.nil* as soon as just half the given list has been disassembled by repeated procedure invocation, requiring just two iterative cycles. By contrast, the program using the *reverse** procedures has to disassemble the entire input list and then match *a.b.c.b.a.nil* with *a.b.c.b.a.nil* in order to terminate, which requires five iterative *reverse** cycles.

The main lesson to be learnt from the palindrome programs shown so far is that in order to secure acceptable computational behaviour with the orthodox term representation of lists, the programming style has become disposed towards more subtle predicates, whose relationships to the original naive, data-independent procedures for *palin* are not trivially perceivable. To prove, for instance, that the *palin** program computes exactly the same relation as the original *palin* program would necessitate a moderate amount of deductive effort. It is useful now to take a contrary stance towards the pursuit of efficient palindrome programs, retaining the original naive *palin* procedures and seeking a data structure representation which allows reasonably efficient computation. (It is assumed throughout this investigation that the intention is to employ Prolog-like control.)

It is useful to present the original *palin* procedures again for further contemplation :-

palin(x) ← *empty(x)*
palin(x) ← *unit-list(x)*
palin(x) ← *first(x,u)*, *last(x,u)*, *middle(x,x')*, *palin(x')*

A data structure representation is required now which allows the first, middle and last components to be directly accessible; here the notion 'directly accessible' means that each required component can be computed by a single invocation of a programmer-defined accessing procedure. The most simple way of arranging this is to employ a 3-ary term of the form *t(u,x',v)* where *u*, *x'* and *v* are respectively the first, middle and last components of the list which the complete term represents. Then the accessing procedures are just three assertions :-

first(t(u,x',v), u) ←
middle(t(u,x',v), x') ←
last(t(u,x',v), v) ←

The question of whether the list *(a,b,c,b,a)* is a palindrome is then posed by the goal ← *palin(t(a,t(b,c.nil,b),a))* . The ensuing computation is then clearly very efficient due to the direct access of the list's components, provided, of course, that the interpreter implements the terms and their matching efficiently. The new data structure representation also improves the behaviour of the palindrome program which tests whether the input list is its own reverse. This is because the *reverse* relation can now be computed using the procedures :-

reverse(nil,nil) ←

reverse(u.nil,u.nil) ←

reverse(t(u,x,v),t(v,x',u)) ← *reverse(x,x')*

which can be implemented iteratively. Note that the orthodox list representation can be retained in order to represent the special cases of the empty list and unit lists. Thus the choice of data structure representation can determine whether recursive procedures are invokable iteratively or only recursively. The t-terms are not suitable for all purposes; in particular there appears to be no way of accessing consecutive pairs in those terms using an iterative procedure, whereas that kind of access is easy when the orthodox terms are used instead.

Unless the interpreter can perform a great deal of compile-time optimization of the source logic program, and so make effective provisions for the storing and accessing of its data structures, the use of terms at run-time is generally unsatisfactory. It is often more satisfactory to use sets of assertions to represent lists, since the mapping of these onto physical memory is then a comparatively easy task for a logic pre-processor. Thus the investigation of palindrome programs turns now to considering the use of assertions to represent the input lists, anticipating computations which merely adjust pointers to array-like representations rather than manipulating cumbersome binding environments induced by the unification of terms.

We consider here just the simplest assertional data structure for lists. This asserts the existence of each member and its index, and asserts the total number of members, that is, the length of the list. For instance, the list *(a,b,c,b,a)* can be named by the constant *L* and then represented by the six assertions :-

item(a,1,L) ← *item(b,4,L)* ← *length(L,5)* ←

item(b,2,L) ← *item(a,5,L)* ←

item(c,3,L) ←

where *length(x,z)* holds when the list *x* has a length *z*. To test whether *x* is a palindrome by accessing its first, middle and last components, the naive procedures for *palin* may be supported by the following rather intimidating set :-

empty-list(x) ← *length(x,0)*

unit-list(x) ← *length(x,1)*

first(x,u) ← *item(u,1,x)*

middle(x,mid(x)) ←

last(x,v) ← *length(x,z)*, *item(v,z,x)*

item(u,i-1,mid(x)) ← *length(x,z)*, *1<i*, *i<z*, *item(u,i,x)*

length(mid(x),z-2) ← *length(x,z)*

The logic can be described informally as follows. The first and last members of the list can be accessed directly by just quoting their appropriate indices. The middle of the list is itself a list, and requires a name to distinguish it from the original list x; the function symbol *mid* is just a naming device which allows us to name the middle of *x* as *mid(x)*. Since *mid(x)* is not just any list, but is uniquely determined by *x*, it is necessary to state in the logic just what it consists of. This is achieved by the last two procedures which can be used to compute all the members and indices associated with *mid(x)* and to compute its length. Hence this formulation of the problem assumes computed access rather than direct access to the middle component. Moreover, the computation is very unsatisfactory when control is wholly top-down, because each time some $mid^n(L)$ is required, this has to be computed by recursively computing the lists $mid^{n-1}(L)$, ..., *mid(mid(L))*, *mid(L)*. What is needed here in order to solve ← *palin(L)* with the procedures and assertions above is an enhancement to Prolog-like control by which the interpreter can compute those lists in a bottom-up manner, allowing each one to overwrite its predecessor once the latter is no longer required for computation. An implementation of this sort would emulate the space-saving economies customarily associated with destructive assignment; iterative bottom-up generation of *mid(L)*, $mid^2(L)$ ... etc. would then be interleaved with top-down execution of the *palin* procedures such that no component was ever accessed more than once.

It would be wrong to conclude from the last example that a top-down interpreter could not satisfactorily interrogate the assertional data structure to test for palindromicity. Logically the problem only requires inspection of the assertions using a simple pointer protocol. Kowalski's method for parsing sentences represented as graphs provides the clue as to how to capture this intention explicitly in the logic.

Introduce a new predicate *palin**(i,j,x)* to express that the fragment of *x* which extends from the i^{th} member to the j^{th} member is a palindrome. It is useful to arrange that *palin**(i,j,x)* holds also when $i \geqslant j$. Then the following procedures are sufficient to solve ← *palin(L)* when *L* is represented by the six assertions given previously :-

palin(x) ← *length(x,z)*, *palin**(1,z,x)*

*palin**(i,j,x)* ← $i \geqslant j$

*palin**(i,j,x)* ← $i<j$, *item(u,i,x)*, *item(u,j,x)*, *palin**(i+1,j-1,x)*

[together with procedures for < and ⩾]

With Prolog-like control (that is, the wholly top-down default strategy), these procedures yield excellent behaviour. The first two arguments of *palin*** serve as pointers which delimit the fragment of *L* about to be inspected for palindromicity. The program can be executed iteratively, induces scarcely any bindings and, provided that some kind of direct addressing mechanism is used to look up the assertions describing *L*, would compete favourably with the execution of a conventional Algol-like program.

Data Abstraction

Data abstraction is concerned with the logical separation of procedures from the concrete data structure representations which they manipulate. This separation has been widely approved in conventional programming as a means of creating clear and flexible high-level procedures; its value lies chiefly in the resulting ease with which both these procedures and the data structures which they will process can be constructed or modified independently and then brought together through the mediation of suitable interfaces. The same motivation prevails in its use in logic programming.

As an example of data abstraction in logic programming, consider the problem of showing that some non-empty finite set *x* is a subset of some set *y*. One possible algorithm is that which, for a set *x* having several members, computes sets x_1 and x_2 satisfying $x = x_1 \cup x_2$, and then investigates the subproblems of showing that both x_1 and x_2 are subsets of *y*. The trivial case is where *x* is a singleton, in which case it is a subset of *y* if its member belongs to *y*. Expressing these ideas straightforwardly in logic produces the procedures :-

$subset(x,y) \leftarrow singleton(x,u),\ u \varepsilon y$

$subset(x,y) \leftarrow union(x_1,x_2,x),\ subset(x_1,y),\ subset(x_2,y)$

where $singleton(x,u)$ expresses $x = \{u\}$ and $union(x_1,x_2,x)$ expresses $x = x_1 \cup x_2$. Now any set of procedures which solves calls to *singleton*, ε and *union* can be regarded as an interface interposed between the *subset* procedures and whatever means are chosen for representing the sets of interest concretely. The procedures above are logically neutral with respect to the concrete representation of the data.

Purely for the sake of example, suppose that terms are used to represent the sets. The use of terms in this way is justified by appealing to existence theorems in set theory. For instance, two such theorems are :-

$(\forall x_1 x_2)(\exists x) union(x_1,x_2,x)$ (every pair of sets forms a union)

$(\forall u)(\exists x) singleton(x,u)$ (every element constructs a singleton)

Using Skolem symbols to instantiate the existentially quantified variables in these theorems, we obtain two Horn clause assertions :-

$union(x_1,x_2,union(x_1,x_2)) \leftarrow$

$singleton(s(u),u) \leftarrow$

which may be treated as procedures serving as an interface between the *subset* procedures and the representation of sets which uses terms constructible from the function symbols *union* and *s*. For instance, the set $\{a,b,c\}$ might then be represented by the term $union(union(s(a),s(b)),s(c))$. Other simple theorems about set membership can be summoned to provide procedures solving calls to ε when sets are represented in this way. Sufficient procedures for ε in the present example would be :-

$u \varepsilon s(u) \leftarrow$

$u \varepsilon union(x_1,x_2) \leftarrow u \varepsilon x_1$

$u \varepsilon union(x_1,x_2) \leftarrow u \varepsilon x_2$

When the interfacing procedures are non-recursive, they may be eliminated from the program by resolving them bottom-up with procedures which call them. Kowalski has interpreted this as the analogue of conventional macroprocessing. It can be regarded as a compile-time transformation (potentially achievable by the interpreter itself, since only resolution is required) which enables the resulting procedures to refer directly to concrete data structure representations instead of having to access them indirectly at run-time by invoking the accessing

procedures. Macroprocessing the *subset* procedures above would result in the new procedures :-

$$subset(s(u),y) \leftarrow u \varepsilon y$$
$$subset(union(x_1,x_2),y) \leftarrow subset(x_1,y),\ subset(x_2,y)$$

When the interfacing procedures are recursive, Horn clause resolution is not generally sufficient to eliminate them from the program. As an example, suppose that sets were represented instead by the more usual terms shown previously, that is, using constructors $\emptyset$ and : . The procedures necessary for accessing these terms for the benefit of the *subset* procedures are as follows :-

$$singleton(u:\emptyset) \leftarrow$$
$$union(u:x_1,x_2,u:x) \leftarrow union(x_1,x_2,x)$$
$$union(\emptyset,y,y) \leftarrow$$
$$u \varepsilon u:x \leftarrow$$
$$u \varepsilon v:x \leftarrow u \varepsilon x$$

Now these cannot be used to eliminate calls to *union* and ε in the *subset* procedures, since they are recursive; resolution would only introduce yet more calls to *union* and ε. Nevertheless it is possible to derive a suitably macroprocessed *subset* procedure set for this term representation, but the necessary inferences use set-theoretic knowledge not present in any of the above procedures. The result of macroprocessing using this knowledge is the procedure set :-

$$subset(u:\emptyset,u:y) \leftarrow$$
$$subset(u:\emptyset,v:y) \leftarrow subset(u:\emptyset,y)$$
$$subset(u:x,y) \leftarrow subset(u:\emptyset,y),\ subset(x,y)$$

The complementary process to macroprocessing is data abstraction. In the procedures below, lists are represented by terms constructed from . and *nil* :-

$$append(nil,y,y) \leftarrow$$
$$append(u.x',y,u.z') \leftarrow append(x',y,z')$$

giving a compact, iteratively computable means of investigating the *append* relation. Suppose, however, that it was required to make the procedures data-independent. Tarnlund (79) has explained informally (attributing the idea to Kowalski) how the terms can be eliminated by introducing new predicates. In the present example, introduce a predicate $append^*(u,x',x)$ which expresses $x = u.x'$. Then the

recursive *append* procedure can be replaced by the pair of procedures :-

append(x,y,z) ← *append*(u,x',x)*, *append*(u,z',z)*, *append(x',y,z')*
append(u,x',u.x')* ←

Similarly the basis procedure can be replaced by the pair of procedures :-

append(x,y,y) ← *empty-list(x)*
empty-list(nil) ←

This transformation has segregated the concrete data from the higher-level procedures. The general rules for achieving this can be summarized as follows. Suppose that some procedure definition contains an *n*-ary predicate $P(T_1, \ldots, T_k, \ldots, T_n)$ where T_1, ... and T_n denote arbitrary terms. Let T_k denote a term of the form $F(S_1, \ldots, S_r)$ where *F* denotes some function symbol and S_1, ... and S_r denote arbitrary terms. Then to eliminate the occurrence of the term T_k from the clause under consideration, introduce a new variable denoted by *X* and a new predicate denoted by *P**. The predicate above is replaced by the predicate $P(T_1, \ldots, X, \ldots, T_n)$ in which *X* has been substituted for T_k, and the predicate $P^*(X, S_1, \ldots, S_r)$ is conjoined to the clause's antecedents. Finally the assertion :-

$$P^*(F(S_1, \ldots, S_r), S_1, \ldots, S_r) \leftarrow$$

is added to the program. This completes the elimination of one occurrence of the function symbol denoted by *F*.

Kowalski (51) has pointed out that the new procedures introduced in data abstraction can enhance the clarity of programs. For example, a further transformation can be conducted upon the procedure definition :-

append(x,y,z) ← *append*(u,x',x)*, *append*(u,z',z)*, *append(x',y,z')*

by abstracting the data-dependent term in its supporting assertion :-

append(u,x',u.x')* ←

This time, however, the dependence of the introduced variable *x* upon *u* and *x'* is expressed by two calls *first(x,u)* and *rest(x,x')* :-

append(u,x',x)* ← *first(x,u)*, *rest(x,x')*
first(u.x',u) ←
rest(u.x',x') ←

In fact any number of calls to selector procedures can be introduced in the abstraction process provided that, conjointly, they establish the correct dependence of the substituted variable upon each of the variables in the term for which it is substituted. The calls to the *append** procedure can now be macroprocessed out to give :-

```
append(x,y,z) ← first(x,u), rest(x,x'),
                first(z,u), rest(z,z'), append(x',y,z')
```

The presence of the *first* and *rest* procedures clarifies the composition of the lists *x* and *z*; it is clear that *x* and *z* have the same first member, and that the rest of *z* is obtained by appending *y* to the rest of *x*. More convincing examples of the stylistic usefulness of data abstraction arise when procedures are required which refer to many distinct data structure components; the calls to selector procedures, if named sensibly by well-chosen mnemonics, serve as program documentation which can also be processed at compile-time to allow efficient run-time access.

CHAPTER 4

REASONING ABOUT LOGIC PROGRAMS

PREVIEW

In order to argue the thesis that logic is a credible programming language it is not sufficient merely to refer to its computational semantics or its amenability to practical implementation. It is also important to show that logic programs can be conveniently subjected to reliable analyses of their logical and behavioural properties. For instance, we may wish to prove that a given program terminates successfully with a correct solution to the problem which it purports to describe.

To articulate such analyses it is clearly necessary to possess some kind of program-reasoning language in which deductions can be made about the relations computed by the programs under examination. The motivation of the present chapter is to establish that this language may be simply predicate logic itself. More precisely, the standard formulation of FOPL can be treated as a general-purpose program-analysis tool which is adequate for specifying programs and inferring their properties. Horn clause logic is then just that subclass of the language which is suitable for computation by virtue of our possession of a convincing procedural semantics and the capability of realizing it in resolution interpreters.

The need for the standard formulation of FOPL is argued from the observation that the facts which we wish to assert during analyses of programs are often not conveniently expressible in clausal form. (It would be pleasing if those facts were so expressible, because resolution would then provide a sufficient inference system for processing them.) Evidence for this argument is offered in the chapter's first section, where it is concluded that standard FOPL provides a more

convincing specification language than clausal form in which to represent the facts of interest about the problem domain. Standard logic therefore plays a central role in such tasks as verification and synthesis of logic programs.

In the next four sections the logic programming formulations of termination, specification, verification and synthesis are introduced, emphasizing their dependence upon non-clausal sentences to capture the program properties of interest. Termination is discussed firstly, because it can be investigated without referring to the notion of an independent specification for a program's computed relation. The method of proof of termination shown here is that developed by Clark and Tarnlund in their pioneering work on logic programming methodology. Logic as a specification language is considered in the third section and contrasted in its applications to logic programming and conventional programming; it is shown that the input-output non-determinism of logic procedure sets allows an interpretation of logic program specification which is more general than that associated with conventional program specification. Here it is also explained why logic programs cannot be usefully treated as self-specifying despite the fact that they explicitly describe the very relations which they compute. The fourth section presents the Clark-Tarnlund treatment of logic program verification which relies not only upon the construction of independent specifications for the computed relations of interest, but also upon inductive characterizations of the data structures manipulated by the programs. This method is of theoretical interest but is unsatisfactory in practice. A more satisfactory approach is to verify programs by deriving them deductively from their specifications. This approach can, of course, also be regarded as a method for logic program synthesis. Its underlying concept is simply that of showing constructively that each procedure definition used in a program is logically implied by an axiomatic description of the relevant problem domain. This derivation methodology is the subject of the final section of the chapter, and underlies all the subsequent material in the thesis.

4.1 : LIMITATIONS OF CLAUSAL FORM

Expression of Computational Problems

Clausal form logic is not always suitable for expressing knowledge about computational problems. This fact is attributable partly to the exclusion of certain logical connectives and partly to the exclusion of explicitly quantified subformulas. Suppose, for instance, that it were desired to express and subsequently solve some problem concerning the membership relation ε holding between elements and sets, the sets being represented by terms constructible from : and ∅. The briefest way of specifying the relation ε of interest is by the non-clausal sentence :-

$$u\varepsilon x \leftrightarrow (\exists v \exists x')(x=v:x',\ (u=v \vee u\varepsilon x'))$$

together with axioms specifying the identity (=) relation. The facts which are summarized by this sentence could be expressed alternatively in clausal form by a conjunction of four clauses as follows :-

$$\leftarrow u\varepsilon\emptyset$$
$$u\varepsilon v:x' \leftarrow u=v$$
$$u\varepsilon v:x' \leftarrow u\varepsilon x'$$
$$u=v \vee u\varepsilon x' \leftarrow u\varepsilon v:x'$$

Here the symbol ∅ is a 0-ary Skolem function representing the empty set, and the first clause above is a consequence of the identity axiom :-

$$\sim(\exists v \exists x')\emptyset=v:x'$$

Now the fourth clause above is clearly a non-Horn clause. Indeed the conversion of most non-trivial sentences of standard FOPL to clausal form gives rise to mixtures of both Horn clauses and non-Horn clauses. Frequently these are quite difficult to interpret, both individually and collectively, particularly when they share several Skolem functions introduced by the elimination of existential quantifiers. A more striking example of this loss of intelligibility is shown in the conversion of the following assertion about unit matrices :-

$$unit\text{-}matrix(x) \leftrightarrow (\forall u)(one(u) \leftarrow ondiag(u,x)),$$
$$(\forall u)(zero(u) \leftarrow offdiag(u,x))$$

This non-clausal sentence, whose import is reasonably clear, converts to the clause set below :-

unit-matrix(x) ← *one(f)*, *zero(g)*

unit-matrix(x) ∨ *ondiag(f,x)* ∨ *offdiag(g,x)* ←

unit-matrix(x) ∨ *ondiag(f,x)* ← *zero(g)*

unit-matrix(x) ∨ *offdiag(g,x)* ← *one(f)*

one(f) ← *ondiag(f,x)*, *unit-matrix(x)*

zero(g) ← *offdiag(g,x)*, *unit-matrix(x)*

which is much more difficult to perceive as a theorem about unit matrices.

Returning to the four clauses describing the ε relation, suppose these are regarded as procedures intended for computational purposes. Then the question of whether or not the fourth clause is a necessary adjunct to the others depends upon the particular problem to be solved by them. If that problem only queries the membership of a given element (and is thus expressible by a goal of the form ← *u*ε*x*) then the second and third clauses alone provide a sufficient procedure set; they would form the Horn clause procedure set customarily used for investigating individuals in the ε relation. By contrast, consider the problem of showing that, for any *u*, the identity *u*=*v* holds if *u* belongs to the singleton {*v*}. Investigation of this problem requires just the first and fourth clauses. Moreover, this investigation will clearly require the use of some proof procedure for general clausal form, and so is beyond the scope of those interpreters which are designed only for executing Horn clause programs. Resolution interpreters for general clausal form are not yet well-developed, although several proof procedures have been examined and proved complete. One such proof procedure is Kowalski's connection graph system (48) which has been implemented (for Horn clause logic only) by Tarnlund (79) at the University of Stockholm. However, it would seem that effective understanding of how best to control non-Horn clause resolution interpreters must await the development of a procedural interpretation capable of satisfactorily explaining the computational significance of mixing top-down and bottom-up inferences. Some simple problems employing non-Horn clause logic are dealt with in Kowalski's report (49), in which he suggests ways of assigning procedural significance to the inferences used there; but these would not be capable of giving a convincing account of the inferences needed for more complicated problems.

Quite often even the goals of problems are not amenable to convenient expression by Horn clauses. An example of such a case is

that mentioned above which seeks to show that $u=v$ holds if u belongs to $\{v\}$. The most natural expression of this goal in logic would be the sentence :-

$$\sim(\forall uv)\,(u=v \leftarrow u\varepsilon v:\emptyset)$$

which is a refutable conjecture asserting that not all instances of u and v satisfying $u\varepsilon\{v\}$ also satisfy $u=v$. This is equivalent to :-

$$(\exists uv)\,(\sim(u=v)\,,\ u\varepsilon v:\emptyset)$$

which in clausal form has to be expressed as two Horn clauses sharing Skolem functions, say f and g, which replace the two existentially quantified variables u and v. This produces a somewhat eccentric rendering of the original goal :-

$$f\varepsilon g:\emptyset \leftarrow$$
$$\leftarrow f=g$$

which asserts that some arbitrary f belongs to some arbitrary $\{g\}$ and simultaneously denies that f and g are identical. These two clauses together with the clauses comprising the ε specification shown earlier then admit a resolution refutation.

The rather inelegant arrangement above suggests that standard FOPL might provide a better external syntax for goals intended for solution by Horn clause procedures, with conversion to clausal form being undertaken by a suitable pre-processor. Then a more natural expression of the goal above in the style of Horn clause notation would be :-

$$\leftarrow (\forall uv)\,(u=v \leftarrow u\varepsilon v:\emptyset)$$

in which the syntax has been elaborated to accommodate non-atomic calls (in this case just one call). However, it is unlikely that such a proposal would significantly enhance the methodology of Horn clause programming, since the kind of problems whose goals benefit from representation in standard FOPL are those whose solutions also require non-Horn clause procedure invocation (as is the case in the present example, which must invoke the fourth procedure for ε in order to derive a refutation). Usually such goals investigate general laws relating whole sets of individuals, and non-Horn clauses are the only practical means of expressing those sets in clausal form. In the current example the sets are the two relations $\{(u,v) \mid u\varepsilon\{v\}\}$ and $\{(u,v) \mid u=v\}$ and the general law expresses inclusion of one within the other. To summarize, then, this example shows that certain computational problems yield a representation in clausal form which is unconvincing in appearance and not suitable for processing by existing interpreters.

Reasoning about Programs

Even when particular computational problems can be expressed satisfactorily by Horn clause programs, reasoning about the logical properties of those programs may not be viable when we attempt to restrict that reasoning to deduction in clausal form logic. The potential utility of deduction for the conduct of such reasoning highlights an important practical distinction between the analyses of logic programs and conventional programs. Reasoning about logic programs essentially consists of making inferences about the relations which they are intended to compute. Reasoning about their run-time behaviour is also of obvious importance, but is perhaps better conceived as reasoning about computations. The semantics of conventional languages do not emphasize this distinction because the intention of a conventional program is to describe a particular computation in considerable detail; the task of inferring facts about the relations which it computes cannot be separated from analysis of its run-time behaviour because both the relations and the behaviour are intimately connected through their dependence upon the mechanism of state transformation applied to the program's variables.

Whereas the logical analysis of a conventional program requires the assumption of an execution mechanism (without which the program has no meaning), the analysis of a logic program only relies upon a suitable axiomatization of relevant knowledge about the problem domain; the program under consideration is just one contribution to this knowledge. Usually it is necessary to summon facts about the problem domain which have no computational utility in themselves yet nevertheless play an important supporting role in the analysis of the program. Whether this is the case for a particular example or not, no assumption of an execution mechanism is necessary in order to infer exactly what the program can or cannot compute. This circumstance allows the attractive possibility of formulating analyses of logic programs as exercises in first order logical deduction which treat logic programs as object-level axiom sets. Thus the tasks of (a) computing individuals in the relations of interest (by execution), and (b) deducing more general properties of the programs which compute those relations (by deduction), closely coincide; this is just the consequence of identifying programs with axiom sets, and computations with deductions.

Some important questions about a logic program which can be investigated by deductive analysis are the following :-

(i) is the goal solvable ?

(ii) do the computed instances of the goal variables satisfy some given specification which is asserted independently of the program ?

(iii) are the program's procedure definitions logical consequences of such a specification ?

These questions address the issues conventionally referred to as (i) proof of successful termination, (ii) proof of partial correctness and (iii) proof of correct synthesis. *[The latter interpretation of question (i) rests upon a relationship between solvability and successful termination which is explained in more detail in the next section.]* It it now recognized that the logical formulation and investigation of such questions about logic programs cannot, in general, be satisfactorily accomplished using just clausal form logic. Proof of termination, for example, usually requires either an independent specification for the program or else an inductive axiomatization of its data structures; in either event the expression of this knowledge requires the construction of non-clausal sentences whose equivalent representations in clausal form would be wholly unmanageable. Likewise, verification and synthesis require the standard formulation of FOPL in order to express their hypotheses in an intelligible way.

The methodological importance of non-clausal logic suggests the need for practical non-resolution inference systems. It seems not too optimistic to conjecture that these may not need to be especially elaborate in order to admit feasible proofs about the properties of logic programs, since both the programs and their specifications just assert object-level facts about the relations of interest. In particular, moderate stylistic restrictions imposed upon the syntax of both programs and specifications may enable some relaxation of constraints such as completeness and hence allow inference systems somewhat simpler than traditional (and complete) natural deduction systems. Moreover, a convincing procedural interpretation of standard FOPL (if we possessed one) might provide useful insights into how best to control such inference systems in order to analyse programs efficiently. But these conjectures can only be tested by empirical investigation of a wide range of 'real-world' programs, and in the present state of knowledge it is not possible to say which methods for reasoning about logic programs will eventually prove to be most practical.

Having outlined the arguments in favour of standard FOPL as a language suitable for reasoning about the logical properties of logic programs, it is now appropriate to consider in greater detail how this language is actually employed for that task. Termination is the first property considered in detail, because it can be discussed without reference to the notion of a program specification; the ideas already presented about logic programs are a sufficient basis for the discussion.

4.2 : TERMINATION OF LOGIC PROGRAMS

The Termination Criterion

An interpretation of familiar ideas like termination and correctness as applied to logic programs was firstly given by Clark and Tarnlund (16). Their termination criterion for a logic program solving a call to some *n*-ary procedure set for a relation *R* is expressed as a conjecture about the existence of individuals in that relation. To make this a little clearer, suppose that the program under consideration is required to compute, as output, instances of some variables $X_{i+1}, \ldots, X_n$ occupying certain argument positions of a call to *R*, given, as input, particular instances $T_1, \ldots, T_i$ in the remaining argument positions. For simplicity of presentation, and without any loss of generality, it can be assumed that $T_1, \ldots, T_i, X_{i+1}, \ldots, X_n$ are respectively associated with argument positions $1, \ldots, n$ in *R*. Then the program's goal will take the form :-

$$\leftarrow R(T_1, \ldots, T_i, X_{i+1}, \ldots, X_n)$$

If no restrictions are placed upon the choice of the input instances $T_1, \ldots, T_i$ then the termination criterion proposed by Clark and Tarnlund is expressed as the requirement that the formula :-

$$(\forall X_1 \ldots \forall X_i)(\exists X_{i+1} \ldots \exists X_n) R(X_1, \ldots, X_n)$$

shall be provable using the program's procedure definitions as axioms. If the above termination formula is provable from those procedures then this guarantees that some satisfying instances of the output variables $X_{i+1}, \ldots, X_n$ exist for any particular choice of the input instances $T_1, \ldots, T_i$. [Obviously the termination criterion has to be reformulated with a different quantification arrangement if a different input-output arrangement in the goal is required.]

Of course, the *execution* of the program also has the object of proving the existence of these satisfying output instances, but it should be noted that successful execution only provides the proof for *one* choice of the input instances (those quoted in the goal). A proof of the Clark-Tarnlund termination formula is therefore more general than a program execution in that it establishes a whole class of goals which are solvable using the procedure definitions.

The justification for interpreting this analysis as a proof of termination is related to the completeness of resolution. For a proof of the termination formula establishes that the program is inconsistent; in which case the completeness of resolution ensures the existence of a refutation derivation, that is, a successfully terminating computation. This means that if the program is executed by a complete resolution interpreter then it must terminate successfully. The treatment of the termination problem for logic programs can be viewed as independent of an execution mechanism in the sense that it is indifferent to the choice of control strategy and hence to the course of run-time behaviour.

The question of whether or not a particular combination of source program and interpreter will jointly determine a terminating computation may be undecidable by the method above if the interpreter's search strategy is such that not all refutations in the search space can be generated. Prolog, for example, will not discover the obvious and immediate refutation for the source program :-

$$\leftarrow set(x)$$
$$set(x) \leftarrow set(u{:}x)$$
$$set(\emptyset) \leftarrow$$

but instead will recurse indefinitely on the first procedure. This is because Prolog assumes by default that the recursive procedure must always be invoked in preference to the basis by virtue of their ordering in the presented text. The termination criterion formulated by Clark and Tarnlund is therefore a criterion for *hypothetical* rather than *actual* termination. It should also be observed that even when the interpreter's search strategy does not discard any refutations, the question of whether or not a given logic program will terminate is only semi-decidable using their method, since if it so happens that the program is consistent (and hence has an unsolvable goal) then an attempt to prove the termination formula must fail; in this event we cannot infer that execution must terminate, since an unsolvable program could well execute indefinitely rather than terminate unsuccessfully. Fundamentally this uncertainty about the behaviour of such a program arises from the undecidability of FOPL.

In his paper with Tarnlund, and elsewhere, Clark has advocated the explicit axiomatization of the data structures which programs are required to process. As well as making assumptions about the data types explicit, this also provides a useful logical basis for either run-time or compile-time type-checking. Additionally, it forms an important part of the Clark-Tarnlund approach to termination proofs.

As an example, consider the problem of symmetrically embedding one palindrome in another to produce a third palindrome. For instance, $x = (c,d,c)$ can be embedded symmetrically in $y = (a,b,b,a)$ to give $z = (a,b,c,d,c,b,a)$. In the previous chapter it was shown that palindromes could be represented concretely by terms of the form $t(u,y,u)$ in which u is any element and y is a palindrome. The empty palindrome is represented by *nil* and a unit palindrome by $u.nil$ where u is any element. A suitable set of embedding procedures is then as follows :-

$$embed(x,nil,x) \leftarrow palin(x)$$
$$embed(x,v.nil,t(v,x,v)) \leftarrow palin(x)$$
$$embed(x,t(u,y,u),t(u,z,u)) \leftarrow palin(x),\ palin(y),\ palin(z),\ embed(x,y,z)$$

supported by the type-checking procedures :-

$$palin(nil) \leftarrow$$
$$palin(v.nil) \leftarrow$$
$$palin(t(u,y,u)) \leftarrow palin(y)$$

The *palin* procedures form part of a first order *Peano*-like axiomatization which inductively generates the class of all terms defined to be of type 'palindrome'. A complete axiomatization also requires, for closure :-

$$palin(y) \rightarrow y=nil \vee (\exists v)y=v.nil \vee (\exists uy')(y=t(u,y',u),\ palin(y'))$$

together with axioms for = over the relevant classes of terms :-

$$t(u',y',u')=t(u,y,u) \leftrightarrow u'=u,\ y'=y$$
$$v'.y'=v.y \leftrightarrow v'=v,\ y'=y$$
$$\sim nil=v.z$$
$$\sim nil=t(u,y,u)$$
$$\sim v.z=t(u,y,u)$$

and an appropriate induction schema for all predicates on the defined class :-

$$(\forall P)(\ (\forall y)P(y) \leftarrow P(nil),\ (\forall v)P(v.nil),\ (\forall uy')(P(t(u,y',u)) \leftarrow P(y'))\ )$$

Now suppose that termination is required for goals of the form $\leftarrow embed(x,P_1,P_2)$ where P_1 and P_2 denote input instances. For example, the actual goal might be :-

$$\leftarrow embed(x,t(a,b.nil,a),t(a,t(b,t(c,nil,c),b),a)$$

which is solved by the instance $x:=t(c,nil,c)$. Then the relevant termination formula for goals of this class is :-

$$(\forall yz)(\ (\exists x)embed(x,y,z) \leftarrow palin(y),\ palin(z))$$

and is required to be provable using the axiom set made up from the procedures for *embed* and *palin* in the intended program together with the axioms characterizing the data structures. Notice that this termination formula is a conditional formula because the instances for the goal's last two arguments are required to be restricted to the type 'palindrome'. If this proof succeeds then the program will terminate successfully with an output instance of x; if the proof fails then termination is undecided.

Proving the Termination Formula

The Clark-Tarnlund method for proving termination begins by instantiating the induction schema in the data axiomatization with an appropriate instance of the termination formula. Using the example above, the instantiation proceeds by replacing every occurrence of some $P(w)$ by the formula :-

$$(\forall z)(\ (\exists x)embed(x,w,z) \leftarrow palin(w),\ palin(z))$$

The result of this is a rather lengthy first order axiom of the form $C \leftarrow A_1,A_2,A_3$ whose consequent and three antecedents are as follows :-

C : $(\forall yz)(\ (\exists x)embed(x,y,z) \leftarrow palin(y),\ palin(z))$

A_1 : $(\forall z)(\ (\exists x)embed(x,nil,z) \leftarrow palin(nil),\ palin(z))$

A_2 : $(\forall vz)(\ (\exists x)embed(x,v.nil,z) \leftarrow palin(v.nil),\ palin(z))$

A_3 : $(\forall uy')(\ (\forall z)(\ (\exists x)embed(x,t(u,y',u),z) \leftarrow palin(t(u,y',u)),\ palin(z)\ )$

$\leftarrow (\forall z)(\ (\exists x)embed(x,y',z) \leftarrow palin(y'),\ palin(z))\ )$

It is required to show that the above axiom $C \leftarrow A_1,A_2,A_3$ together with the Horn clause procedures for *embed* and *palin* logically imply the consequent formula C, which is equivalent to showing that C is provable from the given axioms. This is accomplished by showing that the Horn clause procedures logically imply A_1, A_2 and A_3, which

presents little difficulty. For example, the first Horn clause procedure for *embed* implies :-

$$(\forall z)(\ (\exists x)\mathit{embed}(x,\mathit{nil},z) \leftarrow \mathit{palin}(z))$$

(assuming that $(\forall z)$ ranges over a non-empty universe), and this together with the assertion $\mathit{palin}(\mathit{nil}) \leftarrow$ implies :-

$$(\forall z)(\ (\exists x)\mathit{embed}(x,\mathit{nil},z) \leftarrow \mathit{palin}(\mathit{nil}),\ \mathit{palin}(z))$$

which is just A_1. Proof of A_2 is similarly trivial, whilst the most laborious proof is that of A_3; none of the proofs of A_1, A_2 and A_3 encounters any conceptual difficulties. Proof of C is now immediate.

Proving termination of a logic program in the manner above clearly requires the resources of standard FOPL in order to express concisely both the termination formula and the induction axiom, as well as the sentences making up the proof itself. Although the proof may be quite easy it will normally require inference rules rather more elaborate than resolution in order to be practical. This observation confirms the need for convenient inference rules for manipulating standard FOPL formulations of program properties.

Later on it will be shown that logic program termination can be investigated by a rather different method from that shown above, although the basic notion of proving the termination formula remains the prime objective; rather it is the axiom set underlying the proof which is different in the alternative approach, employing axioms comprising a *specification* in place of axioms comprising executable *procedures*. However, the concept of a specification also underlies the tasks of verifying and synthesizing logic programs and so, being of central importance to several strands of logic programming methodology, is now discussed in a new section.

4.3 : SPECIFICATION OF LOGIC PROGRAMS

Logic as a Specification Language

The use of logic for specifying programs was first demonstrated rigorously by Floyd (25), who attributed the underlying ideas of his use of logic to Perlis and Gorn. The potential contribution of logic to both the theory and practice of computer programming had also been previously recognized by McCarthy (59). At the present time, FOPL is frequently used by computer scientists to specify formally the properties

of programs; yet programmers as a whole make little use of logic, being unconvinced of its efficacy for significant programs. Reservations about logic as a specification language are in any case occasionally expressed by computer scientists themselves. Liskov and Zilles (54), for instance, suggest that axiomatic specifications for significant problems will be inherently incomprehensible and difficult to compose, although they do not offer evidence for this view. Noonan (68) also asserts that logic is impractical for specifying programs which manipulate non-trivial data structures; instead of logic he chooses a *BNF* grammar notation in order to specify a simple parsing problem, but achieves a much less satisfactory result than Kowalski's treatment (49) of parsing using Horn clause logic.

The objections to logic raised by the authors cited above appear to focus on alleged shortcomings in matters of *style*, rather than on questions of whether or not logic is *theoretically* capable of specifying all programs. Their objections can be countered by adopting logical styles which are less formal in appearance and more imaginative in expressiveness than the styles often employed in the objectors' own examples.

More serious criticisms of logic as a specification language have been advanced by Hewitt, McDermott and others, who consider it unsuitable for specifying programs which behave as though they were modifying their own specifications. These objectors have in mind the kind of program typically used by researchers in artificial intelligence where the program's logical competence is governed by an axiomatic data base susceptible to modification as execution proceeds; altering the data base may alter the universes of facts which the program can and cannot prove and refute, so that the notion of an invariant specification here seems to have little utility. This problem of logic's 'monotonicity' will require much investigation before its seriousness can be properly assessed; but it is certainly an inconsequential problem for most mundane programming purposes at the present time.

The Meaning of Logic Program Specification

The way in which a conventional program (by which is meant an Algol-like program) is typically specified in logic is rather less general than the way in which a logic program will be specified. When a conventional program is specified with the usual intention of establishing a proof of correctness, the specification expresses a

requirement that, if execution begins with some predicate $I(A_1, \ldots, A_m)$ holding upon the initial states A_1, ... and A_m of some input variables X_1, ... and X_m, and subsequently terminates, then a predicate $R(A_1, \ldots, A_n)$ will hold where A_{m+1}, ... and A_n are the final states of output variables X_{m+1}, ... and X_n. If the program satisfies this requirement then it is said to be partially correct with respect to I and R. Proving partial correctness requires a preliminary axiomatization of all those program statements which may influence, directly or indirectly, the final states of X_{m+1}, ... and X_n, followed by a proof based upon that axiomatization that the input-output relation described above will be satisfied. Thus the conventional program-proving paradigm confines its analysis to one input-output arrangement for the arguments of the computed relation R.

Specification in logic programming refers to the axiomatic definition of the computed relation R which some procedure set for R is required to compute, and does not assume any particular arrangement of input and output for the arguments of that relation. Whereas successful execution of a conventional program as described above computes instances A_{m+1}, ... and A_n from input instances A_1, ... and A_m so as to satisfy the predicate $R(A_1, \ldots, A_m, A_{m+1}, \ldots, A_n)$, execution of the analogous logic program solves the particular goal :-

$$\leftarrow R(A_1, \ldots, A_m, X_{m+1}, \ldots, X_n)$$

by invoking procedure definitions capable of computing n-tuples of R.

The logic programming analogue of the imposition of an explicit input condition $I(A_1, \ldots, A_m)$ could be a procedure call to I included in the goal as follows :-

$$\leftarrow I(A_1, \ldots, A_m), R(A_1, \ldots, A_m, X_{m+1}, \ldots, X_n)$$

whose execution will firstly invoke procedures for I to verify the input instances and then initiate the investigation of the principal relation R. Alternatively, a call to I could be incorporated in each of the procedures for R, thus restricting the class of n-tuples which they were able to compute. These alternatives correspond respectively to performing input-checking once before initiating the main computation and performing it only at the times when it is immediately needed for computing individuals in R. Robert Kowalski has pointed out yet a third possibility of investigating the computed relation R contingent upon an assumed input condition I. Instead of arranging that calls to I occur in the program text, the program property of interest is now expressed by the formula :-

$$(R(A_1, \ldots, A_m, X_{m+1}, \ldots, X_n) \leftarrow I(A_1, \ldots, A_m))$$

Then the assumed input condition appears in *theorems* about the program rather than in the program itself.

Irrespective of whether or not some input condition is incorporated in a logic program, specification in logic programming is concerned solely with defining a relation R by some axiom set S, with the understanding that any program's set P of procedure definitions *conforms* to S (or, equivalently, is *partially correct* with respect to S for relation R) if and only if every n-tuple computable from them solving the goal $\leftarrow R$ does indeed belong to R as defined by S. It must be emphasized that specification in this sense is therefore associated with a set P of procedure definitions rather than with an entire (goal-containing) program. This distinguishes our approach from conventional program specification which is confined to deal with just one particular choice of input and output variables. Our approach benefits from the fact that, once the set P of procedure definitions is known to be (partially) correct, this knowledge is unaffected by the subsequent choice of goal; S then specifies a whole class of programs exhausting all 2^n possible permutations of the goal's input-output arrangement. This is just a consequence of the input-output non-determinism of logic procedures.

The Need for Independent Specifications

The underlying motivation of proving logic programs to be correct in the sense described above is to ensure that they truly capture our computational intentions. Of course, many simple logic programs can be regarded as self-evidently correct in that they describe the relations which they compute more plausibly than any other descriptions which we could formulate, other than by explicit enumeration of all the individuals in those relations. However, for most non-trivial programs it is not easy to immediately perceive by inspection of their texts that they do indeed correctly formulate our intuitive understanding of the problems of interest. In such circumstances it is necessary to construct a most-plausible specification and then decide whether a given program conforms to it. The need for such a specification prevails in logic programming to no less an extent than it does in conventional programming, notwithstanding the fact that logic programs possess an extremely useful declarative interpretation. For this reason there is little utility in the idea that logic programs might be 'self-specifying'.

Despite the argument above, there remains the interesting question of whether a given program - or, more precisely, a given set of procedure definitions - might usefully be regarded as a plausible specification for some other program, possibly more subtle in its logic, intended to compute the same relation. Consider, for example, the procedure set below, in which we identify the relation *R* with the relation named by the predicate *reverse* :-

reverse(nil,nil) ←
reverse(u.x,y) ← *reverse(x,z)*, *append(z,u.nil,y)*

append(nil,y,y) ←
append(v.z,w,v.y) ← *append(z,w,y)*

Let *P* denote this set of procedures. Then *P* could be said to implicitly 'define' the relation $R^* = \{(x,y) \mid P \vdash reverse(x,y)\}$, which is the set of all 2-tuples computable from *P* by making calls to *reverse*. This set might be regarded as a self-evidently correct specification of the relation *reverse* holding between two mutually reversed lists, in which case we would identify R^* with *reverse*. Now consider another procedure set *P'* capable of solving calls to *reverse* :-

reverse(x,y) ← *append(x,y,z)*, *palin(z)*,
append(y,x,z'), *palin(z')*

append(nil,z,z) ←
append(v.x,y,v.z) ← *append(x,y,z)*

palin(nil) ←
palin(u.nil) ←
palin(u.x) ← *append(x',u.nil,x)*, *palin(x')*

Implicitly *P'* defines the relation $R^{**} = \{(x,y) \mid P' \vdash reverse(x,y)\}$, which is the set of all 2-tuples computable from *P'* by making calls to *reverse*. An interesting question now arises as to what relationship obtains between R^* and R^{**}. In particular, if *P* is intended as a specification for *P'* then we require that R^* shall include R^{**}, in order that any 2-tuple computable from the specified procedure set *P'* shall belong to the relation specified by *P*. In fact it can be shown that R^* and R^{**} are identical.

It would be pleasing to be able to investigate the relationship between R^* and R^{**} above using just object-level deduction, treating the procedures in *P* and *P'* as axioms for this purpose. Unfortunately

it can be shown that this axiom set would not be sufficient in that respect, because *P* does not comprise a complete object-level definition of the *reverse* relation; this fact holds even though *P* is capable of computing every 2-tuple in that relation. As an example of the insufficiency of *P*, observe that it does not admit any object-level deduction which decides whether or not the 2-tuple *(a.nil,nil)* belongs to *reverse*. *P* implies neither *reverse(a.nil,nil)* nor ∿*reverse(a.nil,nil)*. This shows that *P* does not define at object-level the exact membership of the relation whose individuals it is capable of computing. Hence object-level deduction will not be sufficient to decide whether or not an arbitrary 2-tuple, such as might be computed from *P'*, belongs to the relation specified by *P*. *P* is therefore not an adequate specification for *P'* if the relationships between them are to be investigated using object-level deduction only. A proof that *R** and *R*** are equal can be deduced at meta-level by formulating meta-theorems expressing what *P* and *P'* are capable of computing.

Establishing whether or not a logic program conforms to a specification is just one of many tasks to do with reasoning about programs which we would like to pursue by logical deduction. The insufficiency of an axiom set like *P* above obstructs these tasks as well as verification. Consider, for example, the question of whether or not there is any computational redundancy in the behaviour of a program having the goal :-

$$\leftarrow reverse(x,y),\ reverse(y,x)$$

Suppose that *P* is treated as the only knowledge available about the *reverse* relation. In that case there is no object-level deduction using *P* as the sole axiom set which establishes that *reverse(x,y)* and *reverse(y,x)* are equivalent; that is, there is no deduction which shows that *P* implies the following theorem about *reverse*:-

$$(\forall xy)(reverse(x,y) \leftrightarrow reverse(y,x))$$

By contrast, a *reverse* specification consisting of the axiom set :-

$$reverse(x,y) \leftrightarrow (\exists k)(length(x,k),\ ((\forall uij)(item(u,i,x) \leftrightarrow item(u,j,y)) \leftarrow i+j=k+1))$$

$$(\exists k)\, length(x,k) \leftarrow$$

trivially implies that theorem by simply instantiating *y:=x* and then invoking the fact that every list *x* has a length *k*. Then a further trivial deduction shows that the goal above is equivalent to $\leftarrow reverse(x,y)$.

The significant feature of the *reverse* specification just presented is that it contains an if-and-only-if definition of the predicate *reverse(x,y)*, and therefore contains more information about the *reverse* relation than is provided by *P*. For instance, assuming that the meanings of *item* and *length* were also specified for terms constructed from . and *nil*, it would be adequate for deducing at object-level that the 2-tuple *(a.nil,nil)* did not belong to *reverse*. Moreover, given also suitable specifications for *append* and *palin* it would be adequate for deducing the partial correctness of *P'* for the *reverse* relation. A central feature of logic specifications, then, is that by the use of if-and-only-if definitions of the specified relations they assert more object-level knowledge about those relations than do the conforming programs which compute those relations.

Specification Style

Logic program specifications underlie most of the material presented here concerning reasoning about logic programs, and so it is useful to consider now some of their desirable properties other than the minimal technical requirement of completely defining the relations of interest. Briefly, we require that they should be unbiased towards particular kinds of computational behaviour, should employ as little recursion as possible and should allow the definitions of subsidiary relations to be specified separately from the definition of the primary relation of interest. These features are considered in turn.

Non-computational Disposition

Since a specification is naturally regarded as the most authoritative knowledge about the relation under consideration, it is important that it should be sufficiently clear in its import to be treated as self-evidently correct. Its correctness may, of course, be more evident to some persons than to others, since clarity is a subjective matter and depends upon personal intuitions. Nevertheless it seems reasonable to require that specifications should be free of idiosyncratic features which anticipate special behavioural properties of programs designed to conform to them, since such arrangements tend to obscure essential logical content. Generally, then, the style of axioms comprising specifications is disposed towards naive declarative assertions about the problem domain which take no account

of specific algorithms known to be effective for that domain.

An immediate consequence of this recommendation is that specifications and programs conforming to them will not usually share close logical proximity, in the sense that the task of showing that they both deal with the same relation may be a considerable undertaking of logical deduction. The reasons for this are two-fold. Firstly, there may exist no known algorithms which are effective for the problem except those which exploit comparatively 'deep' theorems about the problem domain. Secondly, even if a useful algorithm only exploits theorems easily deducible from the specification, it may be the case that the limitations of the interpreter's control strategy demand the use of rather subtle logical representations of those theorems such as were exhibited in the earlier chapter dealing with logic programming style.

Consider as an example the problem of showing that some given element u is the minimum member of a given set x. Representing the sets in question by terms constructible from : and $\emptyset$, the required *min* relation can be specified with reasonable clarity by the axioms :-

$$min(u,x) \leftrightarrow u \in x,\ lowerbound(u,x)$$
$$lowerbound(u,x) \leftrightarrow (\forall v)(\ u \leqslant v \leftarrow v \in x\ )$$
$$u \in x \leftrightarrow (\exists v x')(\ x = v{:}x',\ (u = v \vee u \in x')\ )$$

where $min(u,x)$ holds if u is the minimum member in x and $lowerbound(u,x)$ holds if u is a lowerbound for x. The membership relation over the chosen class of terms has to be defined recursively in the third axiom. These axioms about the problem domain jointly constitute a specification for procedure sets intended for solving calls to *min*. The elementary relations = and $\leqslant$ are assumed to be implicitly axiomatized, and any calls to them which might appear in conforming procedures are assumed to be directly executable by the interpreter. Now the simplest procedure set which conforms to the specification and employs no predicates other than those already introduced is :-

$$min(u,x) \leftarrow u \in x,\ lowerbound(u,x)$$
$$u \in u{:}x' \leftarrow$$
$$u \in v{:}x' \leftarrow u \in x'$$
$$lowerbound(u,\emptyset) \leftarrow$$
$$lowerbound(u,v{:}x') \leftarrow u \leqslant v,\ lowerbound(u,x')$$

Each of the procedures above asserts a quite obvious fact about the problem domain and can be shown to conform to the specification by

pursuing some quite trivial deductions. Now in practice this is not the procedure set which we would employ with a Prolog-like interpreter to deal with all possible calls to *min*. If both calling arguments are given as input then the computation is quite satisfactory and behaves as a sequence of two essentially iterative tests - the first to confirm membership of the given element and the second to confirm the element as a lower bound. Suppose instead that the procedures were used to *discover* the minimum of a given set. Then the computation would, in general, be very inefficient, since each time some member selected from the set was shown not to be a lower bound, the ensuing backtracking would discard all the comparisons made between that member and the others even though those comparisons could assist subsequent computation.

A more subtle procedure set for solving calls to *min* is as follows :-

$$min(u,u:\emptyset) \leftarrow$$
$$min(u,v:w:x') \leftarrow v<w,\ min(u,v:x')$$
$$min(u,v:w:x') \leftarrow w<v,\ min(u,w:x')$$

These procedures give excellent behaviour with Prolog-like control for both discovery and confirmation of minima of given sets, yet are not immediately obvious consequences of the specification set. To deduce that this procedure set conforms to the specification the transitive property of < has to be exploited in a not altogether trivial way and results in some quite untidy proofs. However, this does not detract in any way from the choice of specification; the task of verifying useful but subtle programs using naive specifications, difficult as this may be, must be viewed as a natural part of the programming process.

Non-recursiveness

Another desirable feature of specification style is the minimal use of recursiveness. Recursiveness cannot always be avoided because some relations are only capable of inductive definition. The ϵ relation over terms used in the *min* example above is such an instance. The objection to recursiveness in an axiom specifying some relation is that its definiens is not completely comprehensible until the axiom as a whole has been understood, which just re-invokes the same difficulty.

In cases where recursiveness is unavoidable it is useful to confine it, if possible, to the axioms specifying the more primitive relations in the specification set. Thus, in the *min* example the *min* and *lowerbound* relations were defined non-recursively, whilst the primitive ε relation employed in their definitions was the only recursively defined relation. As a further example, consider the *subset* specification below which holds over sets represented by terms :-

$$subset(x,y) \leftrightarrow x=\emptyset \vee (\exists ux')(\ x=u:x',\ u\varepsilon y,\ subset(x',y)\ )$$
$$u\varepsilon x \leftrightarrow (\exists vx')(\ x=v:x',\ (u=v \vee u\varepsilon x')\ )$$

These sentences do not provide an especially convincing account of the *subset* relation, and are more appropriately regarded as a computational description of how to solve a call to *subset* using an incremental algorithm; the first sentence clearly anticipates the familiar procedures for *subset* :-

$$subset(\emptyset,y) \leftarrow$$
$$subset(u:x',y) \leftarrow u\varepsilon y,\ subset(x',y)$$

The more natural specification of *subset* replaces that first sentence with a non-recursive one :-

$$subset(x,y) \leftrightarrow (\forall u)(\ u\varepsilon y \leftarrow u\varepsilon x\ )$$

which captures the essential meaning of *subset* in the least obscure way. The preference for the non-recursive *subset* definition is not only an aesthetic one. It so happens that the recursive alternative contains less object-level information than the latter even though it trivially implies useful *subset* procedures. In some investigations of set properties it has to be augmented by an induction schema for the class of terms constructible from : and $\emptyset$; there are several examples in the paper by Clark and Tarnlund (16) in which an induction principle has to be summoned in order to strengthen recursive specifications like that above for *subset*.

Use of Primitive Relations

A logic specification S for some relation R will be permitted to define R relative to some set of primitive relations whose own specifications are not contained in S . More precisely, if S is an axiom set comprising a specification for R and refers to some other

relation R' whose own membership is not determined by S, then R' is said to be a primitive of S. This does not preclude the possibility that S might assert some general properties of R'. The specification recently considered for the *min* problem contained examples of such primitives, namely = and $\leqslant$. The memberships of *min*, *lowerbound* and ε are undetermined in the specification set S :-

$$min(u,x) \leftrightarrow u\varepsilon x,\ lowerbound(u,x)$$
$$lowerbound(u,x) \leftrightarrow (\forall v)(\ u\leqslant v \leftarrow v\varepsilon x\ )$$
$$u\varepsilon x \leftrightarrow (\exists vx')(\ x=v:x',\ (u=v \vee u\varepsilon x')\ )$$

because the memberships of = and $\leqslant$ are also undetermined by S. However we may say that S specifies *min* relative to = and $\leqslant$ in the sense that the meaning of *min* is dependent upon whatever meaning is assigned to those primitives.

In composing a specification set, then, it is not insisted that sufficient information shall be included in it to determine a non-empty denotation for the specified relation. [The denotation of n-ary R in S is the set $\{(x_1, \ldots, x_n) \mid S \vdash R(x_1, \ldots, x_n)\}$.] At this point the reader would be justified in questioning the purpose of allowing a specification for R to define R relative to some set of primitives rather than defining it absolutely; it would seem valid to argue that a proper specification for R should at least determine which members belong to R. The counter-reply to this is that the meanings of the primitives in terms of which R may be defined are sometimes wholly irrelevant to the task of creating or verifying procedures for R. For example, consider the following specification S for Kowalski's *go** relation :-

$$go^*(x,z) \leftrightarrow (go(z) \leftarrow go(x))$$

We might wish to use some procedure set for *go** for the purpose of finding paths in a graph represented by a set of *go** assertions, each one asserting a particular arc in the graph. Now a sufficient procedure set for this purpose is :-

$$go^*(x,x) \leftarrow$$
$$go^*(x,z) \leftarrow go^*(x,y),\ go^*(y,z)$$

which conforms to S *independently* of the meaning of the specification's primitive *go* relation. So in this example there would be no point in providing knowledge about the members of *go*.

The question of whether or not it is necessary to specify particular meanings for the primitives in some specification depends entirely upon the task to which the specification is applied. For instance, the above set of three axioms respectively defining the relations *min*, *lowerbound* and ε, together with some implicit properties of =, is sufficient to deduce the partial correctness of the *min*, *lowerbound* and ε procedures used for the naive *min* algorithm. This holds irrespective of whatever meaning might be attached to the primitive ⩽. On the other hand, the set of three *min* procedures for the more subtle algorithm cannot be deduced to conform unless the specification set is extended to include an assertion that ⩽ is transitive; this assertion restricts the meaning of ⩽ in the specification, although it will still be primitive.

It may be useful to summarize this discussion about primitives by simply saying that in order to show that a procedure for R conforms to a specification S , it is not generally necessary that S should determine the membership of R; this is why the meanings of primitives may be inconsequential. It is important to realize that an individual procedure for R just asserts some property of R (for instance, the property of transitivity of *go**), and this property may transcend consideration of the precise membership of R. Note in particular that the computationally useful *go** procedures say nothing about specific individuals in the *go** relation, the choice of which is wholly unconstrained; this is why they are consistent with - and logically independent of - any choice of graph-defining set of *go** assertions.

4.4 : VERIFICATION OF LOGIC PROGRAMS

The Partial Correctness Criterion

The concept of correctness as applied to logic programs was first investigated by Clark and Tarnlund (16). They defined a set P of procedure definitions intended for computing some n-ary relation R to be partially correct with respect to some 'axiomatic definition' A if and only if every n-tuple which they compute to solve any goal ← R satisfies the definiens of R in A. Their 'axiomatic definition' A broadly corresponds to what has been described in earlier discussions herein as a 'specification set'. If the set P of procedure definitions

also satisfies the termination criterion for some class of goals $\leftarrow R$ then it is defined to be totally correct with respect to A for that class.

The style of the Clark-Tarnlund formulation of correctness can be shown by considering the familiar *subset* relation. Suppose that the axiomatic definition A for the *subset* relation over terms is as follows :-

$$subset(x,y) \leftrightarrow x=\emptyset \vee (\exists ux')(\ x=u:x',\ u\varepsilon x,\ subset(x',y)\ )$$

$$u\varepsilon x \leftrightarrow (\exists vx')(\ x=v:x',\ (u=v \vee u\varepsilon x')\ )$$

and that the procedure set P whose correctness is in question is :-

$$subset^*(\emptyset,y) \leftarrow$$

$$subset^*(u:x',y) \leftarrow u\varepsilon y,\ subset^*(x',y)$$

$$u\varepsilon u:\emptyset \leftarrow$$

$$u\varepsilon v:y \leftarrow u\varepsilon y$$

Allowing A to be the authoritative specification of the *subset* relation, a proof that any 2-tuple computable from P as a solution of a call to *subset** will indeed belong to *subset* as specified by A is a proof of the partial correctness of P with respect to A. Clark and Kowalski (14) have shown that correctness of P can be expressed and investigated either at object-level or at meta-level. Reading P and A as conjunctions of formulas rather than sets, the requirement at object-level is that of proving that the sentence :-

$$(\forall xy)(\ subset(x,y) \leftarrow subset^*(x,y)\ ) \leftarrow P,A$$

is valid. This sentence can be interpreted as the analogue of the 'verification condition' in conventional program proving. It ought to be mentioned here that in an unpublished report by Clark and Kowalski a discussion is given of another example (a verification of a *quick-sort* program) which suggests that they would prefer to have $\leftrightarrow$ in place of the left-most $\leftarrow$ in the above verification condition for the *subset** procedure set. However, that stronger requirement would not be necessary in order just to establish the partial correctness of P. The significance of using $\leftrightarrow$ instead of $\leftarrow$ is that the sentence will then require that a consequence of P and A is that any 2-tuple in the *subset* relation as specified by A must be computable from some call to *subset** solved by P ; that is, P,A will imply that the specified relation $\{(x,y) \mid A \vdash subset(x,y)\}$ is *equal* to - rather than merely *includes* - the computed relation $\{(x,y) \mid P \vdash subset^*(x,y)\}$.

When this is so, the two *subset** procedures constitute a *complete* procedure set for the *subset** relation as specified by *A* through the identification of *subset** with *subset*; the set is complete in the sense that, when placed in union with a similarly complete procedure set for ε, the result (*P*) is capable of computing all individuals in the specified relation.

A somewhat neater expression of the partial correctness criterion is given in the meta-language. The criterion here is that the sentence :-

$$(\forall xy)\,(\,(A \vdash subset^*(x,y)) \leftarrow (P \vdash subset(x,y))$$

should be a meta-theorem, in which case *P* is partially correct with respect to *A*. Again, one can strengthen this criterion by replacing the ← connective by ↔ , which then adds the requirement that *P* is complete with respect to *A*; but this property is not necessary for partial correctness.

Proving the Partial Correctness Criterion

Two quite distinct approaches to logic program verification have so far been researched and published in detail. These can be distinguished informally by saying that the first one treats the procedure definitions and the sentences specifying the computed relation as axioms and pursues a proof of the object-level verification condition as a target theorem, whereas the second one seeks to prove the procedure definitions as target theorems using just the specification as the axiom set. Both approaches generate object-level proofs which establish that the procedure set satisfies its partial correctness criterion for the given specification.

The first approach is demonstrated in the Clark-Tarnlund paper which presents object-level partial correctness proofs for two logic programs (*ordered-tree-insertion* and *quick-sort*). As with their treatment of termination proofs, they appeal to an induction schema associated with data structure axiomatization in order to prove the relevant object-level verification conditions. Proofs of the corresponding meta-level verification conditions do not appear to have been published, although Clark and Kowalski have affirmed that these are structurally similar to those employed at object-level other than in the way they invoke induction principles.

Considering again the *subset* example examined above, object-level proof of partial correctness proceeds quite easily by exploiting the following induction schema for the class of terms chosen to represent sets :-

$$(\forall P)(\ (\forall x)P(x) \leftarrow P(\emptyset),\ (\forall vx')(\ P(v{:}x') \leftarrow P(x')\ )\ )$$

The appropriate instance of P chosen for the proof is :-

$$P(x) := (\forall y)(\ subset(x,y) \leftarrow subset^*(x,y)\ )$$

and the general structure of the ensuing proof of the object-level verification condition :-

$$(\forall xy)(\ subset(x,y) \leftarrow subset^*(x,y)\ ) \leftarrow P,A$$

is then very similar to the inductive termination proofs, using the *subset** procedures, the *subset* specification and the instantiated schema as axioms.

The inductive proofs of termination and partial correctness developed by Clark and Tarnlund have the disadvantage of becoming extremely cumbersome for non-trivial examples. Furthermore it is easy to find cases which do not seem to fall naturally within the scope of their method. For instance, the verification condition :-

$$(\forall xz)(\ \hat{go}^*(x,z) \leftarrow go^*(x,z)\ ) \leftarrow P,A$$

cannot be proved at object-level from the axioms :-

$$P :\quad go^*(x,x) \leftarrow$$
$$\qquad go^*(x,z) \leftarrow go^*(x,y),\ go^*(y,z)$$

$$A :\quad \hat{go}^*(x,z) \leftrightarrow (go(z) \leftarrow go(x))$$

even though *A* trivially implies the reflexivity and transitivity of $\hat{go}^*$. Here there appears to be no obviously useful way of strengthening the axioms with an inductive characterization (constraining the *go* relation, for instance) for the data structures (graphs) to which *P* will be applied.

Quite apart from these failings, the inductive approach to the verification of logic programs fails to provide a satisfactory clarification of the important logical relationships between programs, their specifications and their data structures. In view of this, it is fortunate for logic programming methodology that there exists an alternative way of investigating the correctness criteria for logic programs which :-

(i) dispenses with the cumbersome and slightly confusing role of inductive data structure characterization;

(ii) admits an intuitively more satisfying logical relationship between programs and their specifications;

(iii) allows more manageable proofs of partial correctness;

and (iv) provides an attractive unification of the meanings of verification and synthesis.

This approach was researched independently but contemporaneously by Hogger (38) and by Clark(12), and has since been applied - albeit in differing styles according to its various proponents - for both verification and synthesis of a wide range of programs.

The alternative approach assumes, as before, that a specification set is available to define the n-ary relation R of interest. Let this set now be denoted by S rather than A, since it will be assumed that S possesses the stylistic features of specifications advocated in the previous section. (The 'axiomatic definitions' A typically used in the Clark-Tarnlund paper do not possess these features; on the contrary, they are highly recursive and computationally biased.) Now suppose that T is any sentence logically implied by S, that is, T is a theorem provable using S as an axiom set (these notions being equivalent in FOPL). Then T is a theorem about the problem domain's relations described in S; if S correctly describes the properties of the problem domain, then so does T. In particular, let P be a set of procedure definitions each of which is an example of such a T. If P computes some n-tuple ξ in response to a goal $\leftarrow R$, then we must have $P \vdash R(\xi)$ by virtue of the correctness of resolution (provided that this is realized in the interpreter which computes ξ). In that event we must also have $S \vdash R(\xi)$, because $S \vdash P$ by assumption and $\vdash$ is transitive. Therefore $R(\xi)$ must be a theorem about R, which is to say that ξ belongs to R as specified by S. To summarize this reasoning, it suffices to say that if the procedures in P compute a solution ξ to the goal $\leftarrow R$, and if $S \vdash P$, then ξ belongs to R as specified by S. Now if this holds for *all* n-tuples ξ computable by P in response to any goal $\leftarrow R$, then the partial correctness criterion is clearly satisfied; for this is all that the criterion requires, namely that every computed solution is a specified solution.

It might not be too presumptuous to say that this simple but powerful concept is one of the most important ideas to have emerged so far from research in logic programming methodology, since it not only provides for a satisfactory way of verifying programs, but also gives the means for deriving them.

Termination can also be investigated by using this logical relationship between *S* and *P*. For suppose that the deduction of *P* from *S* is accomplished in such a way as to guarantee that *P* comprises a complete procedure set for *R*. Then all goals $\leftarrow R$ which are solvable using *P* are solvable using *S* and vice versa. Therefore all the computations initiated by those solvable goals must terminate, provided that the interpreter's control strategy does not disallow the generation of any refutations. To establish termination it is sufficient to show that *S* implies the existence of solutions to the goals in question, which may be much easier than the alternative of showing that *P* implies their existence (which is the Clark-Tarnlund method).

To make these ideas more concrete it will now be useful to revisit the *subset* example and examine the correctness proof of the procedure set *P* :-

$$subset(\emptyset,y) \leftarrow$$
$$subset(u{:}x',y) \leftarrow u \in y,\ subset(x',y)$$

using the most natural sentences specifying *subset* over terms :-

$$S: \quad subset(x,y) \leftrightarrow (\forall u)(\ u \in y \leftarrow u \in x\ )$$
$$u \in x \leftrightarrow (\exists v x')(\ x = v{:}x',\ (u = v \vee u \in x')\ )$$

which are presumed to assert self-evidently correct facts about the *subset* and $\in$ relations. It may also be assumed that *S* implicitly contains an axiomatization of =. Now it is very easy to show (by making a definiens substitution for $u \in x$ in the sentence defining *subset* and simplifying the result in two alternative ways) that :-

$$S \vdash (\forall y)\, subset(\emptyset,y) \leftarrow$$
and $$S \vdash (\forall u x' y)\,(subset(u{:}x',y) \leftarrow u \in y,\ subset(x',y))$$

which is sufficient to confirm that *P* is partially correct for *subset* with respect to *S*; that is, every 2-tuple ξ computable from *P* will satisfy $S \vdash subset(\xi)$. Moreover, it is equally easy to deduce from *S* the stronger result :-

$$S \vdash (\forall xy)(subset(x,y) \leftrightarrow x=\emptyset \vee (\exists ux')(x=u:x', u \varepsilon y, subset(x',y)))$$

from which it immediately follows that P exhausts all possible ways of solving a call to *subset*, and is therefore a complete procedure set for *subset*. Now suppose that a complete program body P^* is composed from the union of P with some complete procedure set P' for the ε relation. Then every 2-tuple ξ satisfying $S \vdash subset(\xi)$ will also satisfy $P^* \vdash subset(\xi)$, that is, will be computable from P^* by a successfully terminating computation. If there exists a class of goals $\leftarrow subset(\chi)$ for each of which S implies the existence of at least one solution ξ, then any program consisting of the body P^* and a goal in that class must terminate in execution and hence be totally correct for *subset* as specified by S. Note that a proof of that program's total correctness for *subset* as specified by S does not require investigation of the partial correctness of the assumed procedure set P' for ε - it is only necessary that P' should be complete for ε.

None of the proofs alluded to above for proving the partial correctness of P require the kind of inductions customarily considered essential (in one guise or another) for the axiomatization of classical flowchart programs containing loops. Loops are a major problem in conventional program proving but have no special status in the logic programming formulation of verification. Thus the claim by Reynolds and Yeh (71) that "*induction is the only technique by which programs can be verified*" is not true of logic programs. Manna and Waldinger (60) also conjectured that synthesis of loop-containing programs could not be accomplished without induction, and a similar stance is taken by Spitzen and Wegbreit (78); it will be seen presently that this view, too, does not hold for logic programs.

There would seem to be two facets to a possible explanation of the fact that logic program verifications and syntheses can avoid induction (at least, in all examples examined so far), although it may be that those facets are not mutually independent. Firstly we have the fact that verification which derives procedure definitions from a specification set (as just discussed above) will, in general, use a richer corpus of object-level knowledge for its axiom set than would be the case if the procedures instead were used as axioms for deriving properties of the specification. The latter is the case with the Clark-Tarnlund verification method and is approximately the position

also taken in conventional program verification; in both circumstances it is necessary to appeal to induction principles. Secondly we have the fact that logic programs contain no explicit control information, so that looping behaviour arises from the way in which the interpreter (a) processes recursive procedures and (b) performs iterative search through sets of procedures. The task of showing a procedure to be implied by its specification in a partial correctness proof is wholly indifferent to the way in which that procedure might be eventually executed. For instance, Kowalski's *go** specification trivially implies the recursive *go** procedure which expresses the transitivity of *go** and which, with a typical control strategy, will generate looping behaviour; induction has no role to play here in confirming that the procedure is a true theorem about *go**. Despite these observations, however, we do not yet possess a firm theoretical explanation of what role, if any, induction has to play in the task of reasoning about logic programs.

4.5 : SYNTHESIS OF LOGIC PROGRAMS

Synthesis by Procedure Derivation

In the previous section it was suggested that the notions of verification and synthesis could be unified by the idea of showing procedure definitions to be implied by their specifications. When *S* and *P* are both given, the task of proving *P* from *S* just confirms that *P* is partially correct. However, if *P* is not initially given then that same task can be viewed as a synthesis, in that it results in the formulation of knowledge about the problem domain which was previously unfounded. Synthesis and verification then just connote differing motivations - discovery and confirmation - in the use of *S* to derive knowledge about that domain.

Traditionally the term 'program synthesis' alludes to the task of creating a complete description of an algorithm. This is because the nature of most computational formalisms is such that they provide for the creation of programs whose texts do indeed describe algorithms in almost complete detail. Logic programs, on the other hand, only contribute to the logic components of algorithms, even though their styles may be heavily disposed towards particular control mechanisms. Deductive derivation of procedures from specifications

can be viewed as a process of program synthesis because the derived procedure sets together with arbitrarily chosen goals can be treated as source programs; this is a consequence of possessing interpreters capable of contributing the control components of algorithms.

Because the correctness of derived procedures is independent of control information, their improvement can be pursued solely by the agency of logical deduction. If some given procedure set gives poor behaviour then it may be possible to use it as an axiom set in union with the original specification set in order to derive some new procedures which behave more satisfactorily. Thus the logic programmer who is equipped with a sound and practical inference system for FOPL can pursue both the synthesis and transformation of programs using a methodology which naturally preserves correctness until he eventually finds the optimal program for his intended interpreter. This freedom to develop the logical structure of algorithms using logical deduction and an intuitively correct set of initial axioms describing the problem domain is the outstanding feature of logic program synthesis.

More generally, of course, the systematic incremental development of correct, clear and well-behaved programs has been the central ideal of modern programming methodology ever since Dijkstra's investigations of 'structured programming' (18) and its subsequent refinements by Wirth (89) and many others. However, despite the invaluable improvements which the structured programming era has wrought upon professional programming practice, attempts to fully realize its ideals in the use of conventional programming languages have not satisfactorily surmounted the problem of finding a practical way of associating knowledge about the problem domain with the kind of constructions which those languages typically provide for processing that knowledge. Because those languages do not possess a logical semantics, the programmer's task becomes estranged from the underlying logic of the problem of interest. Just the opposite is true of logic program synthesis; every step the programmer takes towards the development of the program involves him directly in the manipulation of theorems about the problem domain.

Although deduction is a logically sufficient tool for creating logic programs from specifications (or from other programs), this tool requires intelligent control in order to be practical. Whilst the

use of deduction over the problem domain preserves correctness, it offers no guidelines about the computational usefulness of the theorems which it generates. Nevertheless this does not preclude the possibility that some computational intuitions may be reflected in the use of procedural styles imposed upon the presentation of that deduction. We already possess a convincing procedural interpretation of Horn clause logic which allows resolution derivations (which comprise just one kind of deduction) to be viewed computationally; when we are in a position to see that one such computation is more favourable in some respect than another we can then arrange that the control of the interpreter causes it to pursue the more favourable one. Likewise, in the course of deriving procedures from a specification using standard FOPL, it may be useful to have a procedural interpretation applicable to derived sentences which allows them to be assessed in terms of practical merit. For example, given a choice of pursuing derivations from either of two FOPL theorems about the problem domain, a procedural interpretation of those theorems might indicate that one was more likely to lead to practical Horn clause procedures than the other; deciding upon the progress of the derivation on the basis of this assessment is then just another instance of the exercise of intelligent control to guide a deductive formalism.

The ideas expressed above underlie the style in which logic program syntheses are presented here. On the whole these are structured so as to resemble computational derivations; this assists the gradual transformation of wholly declarative theorems (that is, theorems which are almost meaningless from a computational point of view) into Horn clauses whose computational intent is clear. The formulation of logic program syntheses in a quasi-computational style is next explained in some detail.

Derivation as Quasi-Computation

In order to show how procedure derivations may be viewed as computations, it is useful to firstly examine a typical program transformation effected solely by deduction. Consider then the two procedures below which have been selected from the counting program discussed in Chapter 3 :-

count(u.x,w+1) ← delete(u,u.x,y), count(y,w)

delete(u,u.x,y) ← delete(u,x,y)

Now suppose that these are resolved by matching the *delete* literals; the resolvent, which is implied by the two parents, is then the sentence :-

count(u.x,w+1) ← delete(u,x,y), count(y,w)

This can be viewed as a derived procedure for counting the distinct members of list *u.x*. This deduction is a typical logic program transformation which preserves correctness whilst slightly improving computational efficiency. For if the original *count* procedure is replaced by the new one derived above, then solution of some call *count(u.x,w)* can begin by immediately deleting *u* from *x* and counting *y*, instead, as formerly, of firstly invoking the *delete* procedure to delete *u* from *u.x*. In other words, the use of the new procedure eliminates one cycle in the iteration which accomplishes the deletion of all occurrences of *u* from *u.x*.

Now the derivation above can be presented in a goal-oriented style as follows. Suppose that the goal to be solved is ← *count(u.x,w+1)* and that the procedures available for solving it include the two original procedures given above. Then top-down execution of the goal gives the computation :-

← *count(u.x,w+1)*

← *delete(u,u.x,y), count(y,w)*

← *delete(u,x,y), count(y,w)*

At this point in the computation it becomes apparent that the goal ← *count(u.x,w+1)* is solvable if the derived goal ← *delete(u,x,y), count(y,w)* is solvable. From this observation it can be immediately inferred that the procedure :-

count(u.x,w+1) ← delete(u,x,y), count(y,w)

is a consequence of the original procedures. The goal-oriented derivation above is essentially an orthodox top-down logic program execution which has been suspended at some point in order to infer a new procedure, and could be implemented as a compile-time program transformation by appropriately controlling an ordinary resolution interpreter.

A similar derivation style can be formulated for a program synthesis where the axioms comprise a specification set rather than a Horn clause procedure set. In this case, the derivation of a procedure for R likewise proceeds by showing that the goal $\leftarrow R$ is solvable by solving some goal $\leftarrow R_1, \ldots, R_n$ whose calls are all atomic. Frequently the goal which immediately succeeds the initial goal in that derivation has the form $\leftarrow D_1, \ldots, D_m$ whose conjuncts are arbitrary FOPL formulas rather than atoms, in consequence of using standard FOPL to specify R with a sentence :-

$$R \leftrightarrow D_1, \ldots, D_m$$

and then invoking this axiom in response to the initial goal. Clearly if at least one of $D_1, \ldots$ and D_m is non-atomic then the derivation of the desired Horn clause goal will demand inference rules other than resolution, and consequently may not be amenable to such a simple computational interpretation as that afforded to resolution derivations by the procedural interpretation of Horn clause logic.

The computational nature of deductive procedure derivations from specifications treated as input axioms was observed by Clark and Tarnlund (16) in their treatment of verification, although they did not attempt to organize their proofs into the goal-oriented format of typical run-time derivations. (Considering the complexity of their verification conditions it is unlikely that such an attempt would have been very successful.) However, they did observe that some of the inferences could be interpreted as 'symbolic' executions of procedures with generalized arguments rather than arguments instantiated by individuals specific to particular problems. Clark and Tarnlund described such an execution as a 'slow mode' execution, producing generalized procedures which could afterwards be invoked in 'fast mode' to solve specific problems; this suggested the possibility that a logic interpreter might be designed to apply two control strategies - respectively slow and fast modes - to an input specification set together with some goal, such that the first execution provided procedures for solving the goal whilst the second used the procedures to generate the actual solutions. This would obviously be a much more powerful computational tool than existing logic program interpreters.

A significant feature of the counting example above is the fact that the derivation uses only procedures capable of solving calls to *count* and *delete* which are already known to be computationally useful. In other circumstances it is often necessary during derivations to exploit sentences which have no role in 'fast mode' execution (that is, sentences which would not normally be included in an executable program), but which nevertheless contribute important facts about the problem domain. A simple case of this can be found in connection with the *pick* problem which was also discussed in the last chapter. This is the problem of finding two members u and v in a given set z which satisfy $u<v$, in which case the predicate $pick(u,v,z)$ holds. Suppose it is desired to derive procedure sets for solving calls to *pick* when the input sets of interest are represented by terms. A sufficient specification set for the purpose is then :-

$$S \;:\; pick(u,v,z) \leftrightarrow u\varepsilon z,\ v\varepsilon z,\ u<v$$
$$u\varepsilon z \leftrightarrow (\exists vz')(\ z=v:z',\ (u=v \vee u\varepsilon z')\ )$$

Now it is easy to see that S trivially implies the procedures :-

$$P_1 \;:\; pick(u,v,z) \leftarrow u\varepsilon z,\ v\varepsilon z,\ u<v$$
$$P_2 \;:\; u\varepsilon u:z' \leftarrow$$
$$P_3 \;:\; u\varepsilon v:z' \leftarrow u\varepsilon z'$$

These are just the 'if-halves' of the equivalences in S. Note that because, according to the specification, $pick(u,v,z)$ can only be solved by showing $u\varepsilon z,\ v\varepsilon z,\ u<v$ it follows that $\{P_1\}$ is a complete procedure set for *pick*, whilst, by a similar argument, $\{P_2, P_3\}$ is a complete procedure set for ε. Jointly these procedures are sufficient for solving any solvable call to *pick*, albeit rather inefficiently. The ineffiency arises from the execution of a rather large number of superfluous calls to ε. Nevertheless we have in $\{P_1, P_2, P_3\}$ a (partially) correct procedure set which can be used as a sound basis for further derivations even though it has some computational failings. An example is now given to support this assertion.

Consider, then, the consequences of the 'only-if-halves' of the equivalences in S. In particular observe that the *pick* definition trivially implies the procedure-like sentences P_4, P_5 and P_6 :-

$$P_4 \;:\; u\varepsilon z \leftarrow pick(u,v,z)$$
$$P_5 \;:\; v\varepsilon z \leftarrow pick(u,v,z) \qquad P_6 \;:\; u<v \leftarrow pick(u,v,z)$$

These do not form a useful adjunct to $\{P_1, P_2, P_3\}$ for normal computational purposes, yet it will now be shown that they assist the

derivation of a new procedure set which behaves more sensibly than the first one considered. This derivation employs, as an initial axiom set, the theorems in $\{P_1, P_2, P_3, P_4, P_5, P_6\}$ all of which are known to be correct assertions about the problem domain (assuming S is). The goal to which they are applied is that of *pick*-ing u and v from a set $w{:}z'$, and the symbolic execution of this goal proceeds as follows :-

$\leftarrow pick(u,v,w{:}z')$	(initial goal)
$\leftarrow u \varepsilon w{:}z',\ v \varepsilon w{:}z',\ u<v$	(after invoking P_1)
$\leftarrow u \varepsilon z',\ v \varepsilon z',\ u<v$	(after invoking P_3 twice)
$\leftarrow pick(u,v,z')$	(after invoking P_4, P_5 and P_6)

From this a new procedure can be inferred for *pick*; it is :-

$$P_7 : \ pick(u,v,w{:}z') \leftarrow pick(u,v,z')$$

Now this derivation was not deterministic. For instance, the calls to ε in the second goal are processed there in such a way as to ignore the possibility that either u or v is the member w. There exists therefore another derivation which explores an alternative branch from that second goal which deals with the case where u is the first member w. This is depicted below, showing how P_2 - instead of P_3 - is invoked in response to the first call of the second goal :-

$\leftarrow pick(u,v,w{:}z')$	(initial goal)
$\leftarrow u \varepsilon w{:}z',\ v \varepsilon w{:}z',\ u<v$	(after invoking P_1)
$\leftarrow u=w,\ v \varepsilon z',\ u<v$	(after invoking P_2 and P_3)
$\leftarrow v \varepsilon z',\ w<v$	(after simplifying by $u{:=}w$)

The inferred procedure from this derivation is :-

$$P_8 : \ pick(w,v,w{:}z') \leftarrow v \varepsilon z',\ w<v$$

Note that if the first two calls to ε in the second goal are processed respectively by invoking P_3 and P_2 instead of by invoking P_2 and P_3 as shown, then another procedure will be inferred which deals with the only remaining branch from the second goal, namely that for the case where v is the first member w :-

$$P_9 : \ pick(u,w,w{:}z') \leftarrow u \varepsilon z',\ u<w$$

The goal $\leftarrow pick(u,v,w{:}z')$ can now be solved using the procedure set $\{P_7, P_8, P_9, P_2, P_3\}$ instead of the former set $\{P_1, P_2, P_3\}$. This gives a modest improvement in efficiency when Prolog-like control

is used with procedures P_8 and P_9 scheduled at higher priority than P_7. For example, solution of the goal $\leftarrow pick(u,v,4:3:2:1:\emptyset)$ generates just seven ε invocations with this set, whereas twelve are generated in solving that goal with the previous set.

This example has shown how deductive inference is sufficient for firstly deriving a program from its specification and then transforming it to give run-time improvement. In both cases it proved possible to employ a quasi-computational style which allowed the use of orthodox top-down resolution. Resolution is also sufficient for a number of other derivations concerned with the *pick* problem; for instance, if the specification set is extended to include an assertion that < is anti-symmetric (rather than being just any binary relation) then it is possible to show by resolution that P_7 is implied by P_8 and P_9 in a theory which has S as its axiom set, the consequence of which is that P_7 need not be included in the *pick* program. (It is easy to show that in the algorithm suggested above using $\{P_7, P_8, P_9, P_2, P_3\}$ the procedure P_7 is never invoked when the goal is solvable and the relation < is anti-symmetric.)

When specification sets contain sentences which do not trivially imply Horn clause procedures, inferences other than resolution will usually be needed in order to derive the desired procedure set for the problem of interest. In such circumstances it can be helpful to imagine the non-clausal sentences with which they deal as being classifiable in much the same way as Horn clauses :-

$R \leftarrow$	*assertion*
$R \leftarrow R_1, \ldots, R_m$	*conditional assertion*
$\leftarrow R_1, \ldots, R_m$	*denial*

where R, R_1, ... and R_m are arbitrary formulas rather than atoms. Such sentences may then be assigned a procedural interpretation rather like that for Horn clauses. For instance, the sentence :-

$$\leftarrow (\forall uv)(\ u<v \leftarrow consec(u,v,x)\ )$$

is interpreted as a goal with one (non-atomic) call whose arguments are its free variables - in this case, just x. The goal expresses the objective of discovering instances of the variable x satisfying $(\forall uv)(\ u<v \leftarrow consec(u,v,x))$. Likewise, a sentence like :-

$$(\forall uv)(consec(u,v,x) \leftrightarrow consec(u,v,x') \vee (u=u',v=v')) \leftarrow split(x,u',v',x')$$

is interpreted as a procedure having a non-atomic procedure heading with arguments x, u', v', x' and a single atomic call in its body. Pursuing quasi-computations with sentences of this kind in the course of deriving Horn clause procedures from arbitrary FOPL specifications is the central problem confronting the logic programmer who requires formal proof that his programs are correct. The kind of inferences which can be useful for this purpose form the subject of the next chapter. However, the general style of these non-clausal derivations may be appreciated for the brief example below which derives a recursive procedure for the *ord* relation holding upon an ordered list.

Specification Set :

$$ord(x) \leftrightarrow (\forall uv)(\ u<v \leftarrow consec(u,v,x)\ )$$

$$(\forall uv)(consec(u,v,x) \leftrightarrow consec(u,v,x') \vee (u=u',v=v')) \leftarrow split(x,u',v',x')$$

Derivation :

$\leftarrow ord(x)$

$\leftarrow (\forall uv)(\ u<v \leftarrow consec(u,v,x))$

$\leftarrow (\forall uv)(\ u<v \leftarrow consec(u,v,x') \vee (u=u',v=v')),\ split(x,u',v',x')$

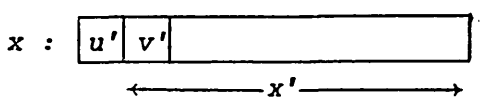
$\leftarrow (\forall uv)(\ u<v \leftarrow u=u',v=v'),\ (\forall uv)(\ u<v \leftarrow consec(u,v,x')),\ split(x,u',v',x')$

$\leftarrow u'<v',\ ord(x'),\ split(x,u',v',x')$

Derived Procedure :

$$ord(x) \leftarrow split(x,u',v',x'),\ u'<v',\ ord(x')$$

The reasoning which underlies the derivation of the successive goals here can be outlined informally as follows : to show that x is ordered, show that all its consecutive pairs (u,v) satisfy $u<v$; but if x can be *split* into components u', v' and x' as depicted below :-



then the set of all consecutive pairs in x is the union of $\{(u',v')\}$ with the set of all consecutive pairs in x'; therefore show that $u'<v'$ and show that x' is ordered subject to x being *split* in this way.

By enriching the specification set above with the further sentences :-

$$(\forall uv)(consec(u,v,x) \leftrightarrow false) \leftarrow empty\text{-}list(x)$$
$$(\forall uv)(consec(u,v,x) \leftrightarrow false) \leftarrow unit\text{-}list(x)$$

other goal-oriented derivations can be pursued to infer two more procedures for *ord* :-

$$ord(x) \leftarrow empty\text{-}list(x)$$
$$ord(x) \leftarrow unit\text{-}list(x)$$

Together the three derived procedures comprise a complete set for the *ord* relation, subject to the assumption that a list can only be either empty, a unit list or a list decomposable by *split*. Each of the procedures is inferred from a branch in a derivation tree explored within a graph of all the derivations determined by the initial specification set S, the presumed inference rules and the initial goal. This consideration supports the view promoted here that procedure derivation and logic program execution are not fundamentally different in principle, but instead just connote particular classes of derivations in FOPL; logic program execution just confines itself to Horn clause logic. Therefore we can see that the two activities investigate problems expressed in a single logical continuum extending from the most restricted subclass of FOPL up to the standard formulation. Consequently there is reason to hope that existing knowledge about the control of program execution might also prove applicable to the strategy of program synthesis using procedure derivation; knowing how best to control a program which investigates the orderedness of a list must provide some insight into the derivation of procedures for investigating orderedness, since our methodology treats these tasks as essentially similar acts of problem solving.

CHAPTER 5

DERIVATION OF LOGIC PROGRAMS

PREVIEW

This chapter describes in detail the more important rules of inference which allow the derivation of logic programs. It is of no consequence here whether this be considered in the context of verification or synthesis because, as explained already, these activities share a common logical foundation.

The first section explains in more detail the motivation and justification behind the proposal that logic procedure derivation can contribute usefully to computer programming. Essentially that proposal advocates just one of the many ways of exploiting the fundamental merit of the logic programming formalism argued in previous chapters, namely the identification of deductive inference as a sufficient device for reasoning about the logical content of algorithms. As well as establishing its logical integrity, we are also naturally concerned that the discipline of logic program derivation should be well-organized from a *logistical* viewpoint, in the sense that its practitioners should be able to deploy their logical resources (axioms and inference rules) in a coherent way. Therefore this section also explains a few general principles which deserve to be incorporated in the derivation methodology.

The inference rules discussed here, which are intended for the gradual transformation of arbitrary FOPL goals into Horn clause goals, fall naturally into two kinds : those which apply structural simplifications to goals independently of the axioms in the specification set, and those which exploit those axioms

in order to combine goals with knowledge about the problem domain. These two classes of inference rules are discussed respectively in two further sections, which conclude by showing how the two kinds of goal transformation are cooperatively interleaved in practice.

The fourth section deals with the derivation of the principal program procedures, that is, procedures not directly concerned with the relatively low-level problems of data access. Some techniques are presented for deriving both recursive procedures and their bases, and the section closes with a discussion of the completeness of procedure sets, which is to say, their capacity to compute all individuals in the specified relation.

The topic of the final section is the derivation of data-accessing procedures, considering in turn the way these can be derived to provide access either to terms or to sets of assertions. A rather interesting example is given there of how to derive the kind of procedures which manipulate explicit pointers in order to systematically process the components of assertional data structures. The material presented altogether in the chapter is then sufficient for the examples of derived programs demonstrated in the final chapters of the thesis.

5.1 : MOTIVATION AND ORGANIZATION OF DERIVATIONS

Motivation

The motivation which underlies the program derivations presented in later chapters is to contribute evidence for the thesis that logic comprises a practical formalism for the creation and expression of computer programs. General acceptance of the view that logic can satisfy this role will depend, not so much upon proofs of its theoretical adequacy for the task (which is not in doubt), as upon its capacity to satisfy pragmatic user-oriented criteria expressing the essential tenets of good programming practice. Important requirements of any formalism for program derivation which is intended to meet such criteria are (a) that it should provide for the explicit expression of the knowledge about the problem domain which is used in the course of program development, (b) that it should allow practical proofs that the resulting programs conform to their specifications and (c) that its application should produce programs which behave

sensibly without necessitating a concomitant loss of logical clarity. These requirements can be met in the derivation of logic procedures by using, respectively, (a) well-styled FOPL specification sets to declare facts about the problem domain, (b) a sound and practical deductive inference system, and (c) good logic programming style.

The motivation expressed above chiefly anticipates the needs of programmers engaged in the task of developing programs without the assistance of 'clever' programming aids. Of course, informal investigations of logic program synthesis of the kind portrayed in this thesis may eventually provide insights into the appropriate construction of new programming tools such as semi-automatic logic program synthesizers, or even interpreters capable of directly executing non-clausal logic as a programming language in its own right. The possibility of gaining such insights is related to the task of finding a comprehensive procedural interpretation for standard FOPL; this we do not yet possess, although the quasi-computational style of the derivations presented here might be regarded as a preliminary step towards that eventual goal. The prospects for partially mechanizing logic procedure derivations, particularly in the light of progress made in other closely related computational formalisms, are considered in the closure of the thesis.

Substantial efforts have, of course, already been made to implement mechanical proof procedures for non-clausal logic. Some of these are discussed in a report by Bledsoe (5). However, these efforts have been predominantly directed to the problem of developing more-or-less autonomous, *intelligent* proof procedures whose implementations could be expected to take over the role of human ingenuity in problem solving. Projects of this kind often reflect the objectives of those researchers in artificial intelligence who regard intelligent general-purpose problem-solving programs as convenient operational representations of some intended theory of intelligence, the latter usually being their ultimate goal; the potential capability of such programs to emulate the role of programmers dealing with specific computational problems in the real world is then somewhat incidental to this ambition.

The attitude underlying the study of procedure derivation in the present work assumes - although this must be a cautious assumption -

that advances in artificial intelligence will not significantly reduce the need for human intuition in problem solving (and hence in computer programming) for some considerable time. This view justifies the pursuit of more modest short-term improvements in the tools with which programmers pursue their computational objectives. In any case, much of the intellectual pleasure of the programming discipline derives directly from the human inventiveness and experimentation currently essential for effective programming, and there seems to be no pressing reason to pursue the demise of this contribution from the programmer; of greater urgency is the need to provide him with tools which allow the clear and accurate expression of his intellectual skills.

Therefore, rather than contributing to the long-term goal of devising autonomous problem-solving systems, the present study focuses instead upon the task of ameliorating the most serious problem afflicting the current practice of computer programming, which is undoubtedly the problem of ensuring correctness. Uncertainty about correctness has its origins in the fact that conventional languages possess no useful declarative semantics, and so do not in themselves provide either for the logical confirmation that programs compute the correct relations,or for the logical derivation of programs from declarative statements about those relations. It does not appear likely at present that conventional program proving will eventually overcome these difficulties : firstly because the notion of retrospective verification is inherently unsatisfactory (due to its somewhat eccentric requirement for the logic of the program to be employed *after* writing the program rather than before - which raises serious questions about how the program was firstly conceived) ; and secondly because the preliminary axiomatizations upon which it depends are made prohibitively complicated by the program's control information, which consists not only of explicit control structures but also of the control implicitly encoded by the use of destructive assignment. Deductive derivation, on the other hand, which encourages the programmer to express and infer knowledge about the problem domain and liberates the soundness of that task from considerations of efficiency, seems to offer much better prospects for overcoming the correctness problem.

Methodological Principles

Since the thesis is centrally concerned with the derivation of procedures from specifications as the basis of logic program synthesis, and since the theoretical justification of the method has been explained in the previous chapter, it is now appropriate to consider some methodological constraints intended to assist the good conduct of derivations. It is also natural to expect the same constraints to prove valuable in the task of program transformation. Whereas synthesis develops new procedures from specifications, transformation develops them from given procedures (and sometimes specifications as well). There is no fundamental distinction involved here, since both activities have the object of deriving Horn clause procedures from whatever logical knowledge is made available; it is only in this initial resource that the two activities differ.

Proposed below, then, are five such constraints imposed upon the task of deriving procedures ; most of them are borrowed from general programming methodology and are just expressions of common sense.

1] Self-evidently Correct Specifications

Showing that a set of logic procedures is implied by a specification has little utility unless the specification correctly summarizes the facts about the problem domain of interest. In formulating the initial axioms for a procedure derivation, we shall abide by the recommendations stated in Section 4.3 stipulating the desirable properties of specifications; these will help to guard against poor specification style, although the potentially difficult task of formalizing the problem domain must always be prone to error. Logic has no answer to this problem, except perhaps as a device for checking the equivalence, or just the consistency, of alternative, independently developed problem formulations.

2] Derivation by Sound Inference

Whilst intuition plays an essential role in guiding the direction of procedure derivation, it is important that the logical expression of the conclusions drawn from that intuition should not itself rely upon that intuition for its correctness. Instead, derivations will be constructed using sound inference rules to ensure that each derived

goal is logically implied by the initial goal and the initial axioms about the problem domain.

3] Goal-oriented Derivation

In order to motivate derivations towards procedures for solving the specific problem at hand, it is desirable that they should proceed (as far as practical) in a top-down goal-directed style in much the same way as programs are developed in other programming methodologies. This reflects the requirement that each step in a derivation should contribute to solving the problem of deriving a useful procedure for the particular relation of interest. This avoids the combinatorial problems associated with bottom-up styles which combine general problem-independent facts in a manner not motivated by specific goals. This principle is therefore based upon arguments similar to those which justify the goal-directed strategies normally used in logic program execution.

4] Logical Clarity in Programs

It is important that derived programs should be reasonably clear in what they assert about the relations of interest, even though the derivation process inherently guarantees their correctness. This is because the opportunities for understanding how to transform those programs or assimilate them into other programs are seriously diminished if their logic is incomprehensible. Unfortunately it is often difficult to find clear logic which also gives excellent behaviour, due to current limitations in implementation technology. Our attitude towards this conflict between logical clarity and efficient behaviour is expressed in the stipulation that, in the course of deriving a set of procedures, the coupling of logic with control will be disposed towards choosing those procedures which stand in greatest logical proximity to their specification and at the same time allow acceptable behaviour with the intended interpreter.

5] Use of Data Structure Abstraction

The last major principle also conforms to a generally accepted tenet of good programming practice, which is that detailed decisions about the implementation of concrete data structures should be deferred until the higher-level procedural properties of the program have been established. This prevents the text of the developing program from

becoming confused by extraneous and procedurally unimportant details of representation. More importantly, it helps to clarify consideration of which aspects of the final algorithm are influenced by the logical properties of the data structures and which aspects depend only upon their concrete run-time implementation; this kind of knowledge is extremely useful when seeking localised modifications - either to logic or to implementation - which improve efficiency. Postponement of the choice of concrete data structure representations can be arranged in the early stages of logic program synthesis by specifying abstract data structures with appropriate sets of selector procedures; near the end of the synthesis, these can be summoned for the purpose of soundly substituting concrete representations, for example by macroprocessing or more sophisticated transformations of the data-accessing parts of the program.

Hierarchical Program Development

The development of a complete program body to solve some goal $\leftarrow R$ can be usefully organized as a hierarchical process in which each level deals with the development of procedure sets serving calls in procedures at higher levels. The highest level just develops a procedure set for R; the lowest levels typically deal with data access. This arrangement improves the management of program composition and also allows a more precise meaning for the term *synthesis* as used in logic programming. A simple example is now outlined to explain this.

Suppose that we require a program body capable of solving calls to *subset*. Then we say that a *procedure derivation for subset* is a single top-down derivation of the kind presented in Section 4.5 from which it is possible to infer a Horn clause *subset* procedure. Moreover, we say that a *synthesis for subset* is the set of all procedure derivations for *subset* which contribute to the final program body. A *complete synthesis for subset* with respect to a given class of calls to *subset* is a synthesis for *subset* which contributes a sufficient procedure set for *subset* for solving every solvable call in that class.

Consider then the pursuit of a synthesis for *subset* with a specification set S containing the axiom :-

$$subset(x,y) \leftrightarrow (\forall u)(\ u \varepsilon y \leftarrow u \varepsilon x\ )$$

It is not possible to derive a practical Horn clause procedure set from

this axiom alone, and so some more information about sets must be added to S . Now suppose we appeal to set theory for knowledge about the constructibility of sets; this will provide the fact that a set is either the empty set or a singleton or the union of two sets. The minimal knowledge which S must contain in order to exploit this fact for the derivation of a procedure set for *subset* is :-

$$(\forall u)(\ u \varepsilon x \leftrightarrow \mathit{false}) \leftarrow \mathit{empty}(x)$$
$$(\forall u)(\ u \varepsilon x \leftrightarrow u = v\ ) \leftarrow \mathit{singleton}(x,v)$$
$$(\forall u)(\ u \varepsilon x \leftrightarrow u \varepsilon x_1 \vee u \varepsilon x_2) \leftarrow \mathit{union}(x_1,x_2,x)$$

The four axioms introduced so far to S together with implicitly assumed properties of = then admit a synthesis for *subset* which consists of three derivations; each derivation contributes one of the three procedures below :-

$$\mathit{subset}(x,y) \leftarrow \mathit{empty}(x)$$
$$\mathit{subset}(x,y) \leftarrow \mathit{singleton}(x,v),\ v \varepsilon y$$
$$\mathit{subset}(x,y) \leftarrow \mathit{union}(x_1,x_2,x),\ \mathit{subset}(x_1,y\ ),\ \mathit{subset}(x_2,y)$$

This establishes the top level in the hierarchical development of the program body.

Inspecting the derived procedures, it is apparent that other procedure sets must also be devised in order to solve their calls to *empty, singleton, union* and ε. All these relations are primitives of the axiom set shown so far, which does not determine their members. It was only necessary to utilize some *consequences* of calls to them in order to derive the procedure set for *subset*. The next level in the hierarchy would now comprise four respective syntheses, each one based upon its own specification set. Yet another level will be needed if any of those syntheses generate procedures containing calls to primitives. Observe that we do not attempt to establish a single global specification set containing sufficient axioms for deriving the entire program body - often these cannot be clearly perceived until the hierarchy has been expanded enough to reveal which relations are interrogated by calls in the derived procedures. This gradual development of specifications as well as procedures around a developing procedure-call hierarchy is rather like a number of conventional program development packages already in commercial use.

Choice of Specification Set

The choice of the initial axioms to comprise a specification set is naturally the crucial determinant of the richness of the class of procedures which can be derived from it. The problem of choosing these axioms is therefore one of the central problems of program synthesis, and is at least as serious as the other notable problem of prescribing an effective strategy for controlling the inference system once suitable axioms have been assembled. *Synthesis* in the fullest sense of the word consists not only of choosing between alternative derivations, but also of choosing between alternative axiomatizations of the problem of interest.

A complete programming methodology would provide techniques for discovering and appraising alternative axiomatizations, given just the computational goal and a minimal description of the relevant problem domain. At present our understanding of the ontological principles which underlie effective algorithms is too rudimentary to allow a very significant step towards such capabilities. Of course, there do already exist implemented synthesizers which can accept minimal problem descriptions and subsequently produce some modestly good, but unsurprising, executable programs as output. Systems of this kind owe their competence to the empirical accumulation of a large data base to which they refer - in a tightly controlled way - in the course of choosing axioms and synthesis rules to deal with the specific problem at hand; yet although their repertoire may seem impressive upon first acquaintance, there does not exist any coherent general theory which justifies their particular *modus operandi*. For instance, although we now possess synthesizers which can output families of sorting algorithms, not one of them is able to pursue those algorithms selectively using criteria of computational efficacy; that is, not one of them will pursue a *quick-sort* algorithm in preference to a *bubble-sort* algorithm by undertaking the intricate analyses required to determine their respective asymptotic comparison counts or other parameters of computational efficiency.

In the syntheses presented here, the requisite axioms are induced by unformalized intuitions concerning the overall structure of the target algorithms. In the absence of firm theoretical - or even empirical - guidelines, those intuitions will have to suffice for our purpose.

With many simple problems rudimentary but acceptable programs can be derived without using any knowledge other than the minimal problem description. The *subset* example just considered approximates to this kind of problem. Suppose the input description is slightly reformulated to express the assumption that the arguments of *subset* have a type called *set*. The primary definition will then be :-

$$subset(x,y) \leftrightarrow (\forall u)(\ u\varepsilon y \leftarrow u\varepsilon x),\ set(x),\ set(y)$$

Then we may appeal to a general problem-independent axiomatization describing what it means for any z to be of type *set* :-

$$\begin{aligned} set(z) \leftrightarrow\ & empty(z) \\ & \vee\ (\exists v)\, singleton(z,v) \\ & \vee\ (\exists z_1 z_2)(set(z_1),\ set(z_2),\ union(z_1,z_2,z)) \end{aligned}$$

$$empty(z) \leftrightarrow (\forall u)(\ u\varepsilon z \leftrightarrow false)$$

$$singleton(z,v) \leftrightarrow (\forall u)(\ u\varepsilon z \leftrightarrow u=v\ )$$

$$union(z_1,z_2,z) \leftrightarrow (\forall u)(\ u\varepsilon z \leftrightarrow u\varepsilon z_1 \vee u\varepsilon z_2\ )$$

This *set* data-type characterization is now no longer an axiom set special to the problem of deriving *subset* procedures; instead it is a corpus of general knowledge which can be summoned to assist derivations for any problems which explicitly express the assumption that they are dealing with objects of type *set*. The set of three *subset* procedures given earlier are derivable quite trivially from the above type-characterized *subset* definition and the type specification together with axioms for =.

It should not be thought that this minimal description of the *subset* problem admits just one feasible procedure set for *subset*. The richness of the class of derivable procedure sets depends also upon the proof procedure employed to search the derivation graph. For instance, the kind of inference rules presented later on are such that, applied to the axioms above, they admit derivations of an alternative procedure set :-

$$\begin{aligned} subset(x,y) \leftarrow\ & empty(x) \\ subset(x,y) \leftarrow\ & v\varepsilon x,\ v\varepsilon y,\ singleton(x_1,v),singleton(y_1,v), \\ & union(x_1,x_2,x),\ union(y_1,y_2,y), \\ & subset(x_2,y_2) \end{aligned}$$

This procedure set schedules the membership tests in a Prolog-like computation quite differently from its predecessor. Also, unlike the latter, the recursive procedure can be invoked iteratively. Thus the intelligent selection of derivations using problem-independent knowledge is sometimes sufficient for obtaining interesting and useful algorithms. In Chapter 7 it will be seen that the simpler sorting algorithms may similarly be derived using no more than a rudimentary definition of sortedness together with some general axioms about lists, sets and ordering relations. It is only for the much deeper algorithms, like Batcher's *merge-exchange-sort*, that the initial specification set has to be furnished with non-trivial pre-proven theorems selected from the mathematical theory of sorting.

Implicit Specification Axioms

On many occasions it is useful to assume that the specification set implicitly contains axioms about elementary relations like = . In many of the derivations presented later on, goals will be transformed in simple ways which depend upon such axioms but which are not explicitly formulated within the derivation texts. For instance, the inference step shown in the derivation below exploits the 1:1 property of = which *S* implicitly asserts :-

$$\vdots$$
$$\leftarrow (\forall u)(\ u \varepsilon y \leftarrow u=v)$$
$$\leftarrow v \varepsilon y$$
$$\vdots$$

This inference is used in the derivation of the *subset* procedure shown earlier which shows that $\{v\}$ is a subset of y by showing that v is a member of y. No attempt is made here to delineate the exact set of assumed axioms about relations like = , but when any rather special property is exploited in an inference step then an informal note about this will be given alongside the derivation. However it can be assumed that *S* will contain at least the knowledge of every tuple in =, every tuple not in =, and the fact that = is a 1:1 equivalence relation.

It is also convenient for presentation's sake to assume some implicit axioms in *S* which allow apparently surreptitious inferences dealing with terms denoting arithmetic expressions. Thus the *kount*

procedure discussed in Chapter 3 :-

$$kount(u.x,w+1) \leftarrow kount(x,w)$$

may be used to infer directly the procedure :-

$$kount(u.x,w) \leftarrow kount(x,w-1)$$

The assumption which justifies this is that the first procedure merely abbreviates the conjunction of the following pair :-

$$kount(u.x,w') \leftarrow kount(x,w), plus(w,1,w')$$
$$plus(w,1,w+1) \leftarrow$$

and that *plus* also satisfies the axiom implicitly contained in S :-

$$plus(w-1,1,w) \leftarrow$$

It should be clear that the original two *kount* procedures shown in the inference step above are both logically implied by the three assumed procedures. These kinds of inference will not generate any semantical difficulties at the superficial level at which they appear in later derivations. Any derivation manipulating terms in the manner above (performing quasi-arithmetic on them, for example) can be justified in first order logic by reformulating it in the way just shown.

Objectives of Derivations

The *logical* objective of a derivation is to show that a Horn clause procedure is logically implied by S . As has been described already, this task is organized as a top-down derivation from the goal $\leftarrow R$ where R is the relation interrogated by the target procedure. The derivation is then a sequence $(\leftarrow R, \ldots, (\leftarrow R_1, \ldots, R_n))$ of goals of which each is logically implied by the conjunction of S with $\leftarrow R$. From the derivation we infer the implication :-

$$S \vdash (R \leftarrow R_1, \ldots, R_n)$$

in order to obtain the target procedure.

The *computational* objective of a set of derivations for R (that is, a synthesis for R) is a set P of Horn clause procedures adequate for investigating some supposed class of goals $\leftarrow R$. If this class contains goals which collectively compute all individuals in R then the objective is a complete procedure set for R. It is presumed, of course, that the adequacy of this set takes cognizance of the efficiency

of the algorithms which it gives with the intended interpreter. A sensible way to proceed is to derive some initial procedure set, investigate its behavioural properties and then resume the derivation process in order to obtain some efficiency-improving transformation. We saw an example of this previously with the *pick* problem.

Inference Rules for Procedure Derivation

The inference rules presented here for procedure derivations are not unlike some of those found in conventional natural deduction systems for FOPL. However, the treatment herein does not appeal to the customary formulations of natural deduction since the latter do not seem to be amenable to a useful procedural interpretation. Instead, the rules presented shortly are intended specifically for the kind of goal transformations which reflect our computational view of procedure derivation. In general they are applied with the intention of inferring from a given goal :-

$$\leftarrow R_1, \ldots, R_k, \ldots, R_m$$

a new goal :-

$$\leftarrow R'_1, \ldots, R'_n$$

by activating some procedure call R_k (not necessarily atomic), and processing it either using just knowledge about the other calls in the goal or else using knowledge invoked from some axiom in the specification set.

The last remark indicates that the inference rules fall into two kinds. A rule of the first kind, called *goal simplification*, typically simplifies the current goal by, for instance, simplifying the activated call's connectives or quantifiers or by finding an instantiation of its variables which trivially solves it. Whatever way is used to simplify the goal, no reference is made to S. A rule of the second kind, called *goal substitution*, often has the opposite properties. It invokes some fact from S in response to the activated call, typically substituting new knowledge into the goal. The skill of logic procedure derivation lies in finding intuitively sensible ways of applying the simplifications and enrichments afforded by the rules in a cooperative manner. When the rules have been discussed in more detail it will be seen their most simple manifestation is in the case where all the manipulated sentences (goals and axioms) are in Horn clause form, in which case they reduce to the mechanisms of an ordinary

top-down resolution interpreter. Their more elaborate manifestations are wholly consequences of admitting the possibility that an activated call may be non-atomic, which in turn ensues from the adoption of standard FOPL for specifying the relations of interest. Whether or not the inferences are applied to non-clausal sentences, they reflect the typical execution of logic programs in that they replace activated calls by conjunctions of other calls and result in the transmission of data between the calls' arguments. However, one important difference between program execution and procedure derivation is that in the former case we have a complete inference system (resolution), whereas we make no claim to possess a correspondingly complete inference system for the latter; the rules examined here are just those which have proved useful in the examples considered. Of course, we would hope to possess eventually a complete, compact and empirically useful inference system together with an intelligent control strategy to govern its application.

The division of the rules for procedure derivation into two kinds is not an arbitrary one, but is rather a useful means of controlling the introduction of knowledge during goal transformation. It would be possible to combine them into a single class, but this would tend to obscure the relative logical contributions made by the current goal and the specification set to the derivation of the succeeding goal; it seems desirable - to allow greatest insight - that these contributions should remain separate in both the concept and the representation of the derivations. In any case, the combined rules would almost certainly be syntactically unmanageable.

Before discussing the various kinds of inference steps in greater detail, it will be useful firstly to announce a small matter of terminology for expressing the logical relationships between goals. It has been pointed out that a rule of the first kind - goal simplification - takes no account of the specification set. If G_r is the current goal and G_{r+1} is the goal derived from it by a rule of this kind, then we shall certainly have the relationship :-

$$G_r \vdash G_{r+1}$$

wherein G_{r+1} is *logically* implied by G_r : which in FOPL is equivalent to saying that G_{r+1} is provable in a theory whose axioms are G_r together with the axioms of FOPL. Moreover, if the relationship below also holds :-

$$G_{r+1} \vdash G_r$$

then G_{r+1} and G_r are *logically* equivalent. Now consider the derivation of G_{r+1} from G_r using a rule of the second kind - goal substitution. The logical relationship between these goals and the specification set is now :-

$$S, G_r \vdash G_{r+1}$$

wherein G_{r+1} is *logically* implied by S,G_r. Now since the existence of S is always assumed in discussing relationships between goals, it is convenient to introduce some terminology which expresses that assumption concisely within such discussions. So when the relation above prevails, we shall say that G_{r+1} is *S-implied* by G_r, and likewise if the relationship below also prevails :-

$$S, G_{r+1} \vdash G_r$$

than we shall say that G_{r+1} and G_r are *S-equivalent*. To say that one goal is *S-implied* by another is just to say that the former is provable in a theory which has as its axioms both S and the latter goal together with the axioms of FOPL.

5.2 : GOAL SIMPLIFICATIONS

This section considers some of the more common ways in which a goal can be simplified in the course of procedure derivation. The discussion is limited to considering this just from a syntactical point of view, although it is possible to assign very general problem-solving interpretations to some of the inference rules. In each case it is assumed that we have some arbitrary goal consisting of a conjunction of calls, and the derivation of the successor goal arises by selecting just one call for simplification.

Deletion of an Implied Call

One of the simplest ways of simplifying the current goal G_r is to delete one of its calls. Since it is assumed here that no sentences other than the current goal are initially given for this purpose, the simplification can only proceed in such a way as to satisfy $G_r \vdash G_{r+1}$ if the deleted call is implied by the other calls. It may be helpful to formalize this in terms of the inference rule *modus tollens* since this

rule underlies our normal mode of top-down goal execution. Suppose that the goal in question is :-

$$G_r : \leftarrow R_1, \ldots, R_k, \ldots, R_n$$

and that the activated call is R_k. Here the calls are labelled so that the assumption that R_k is implied by some conjunction of other calls in the goal is expressible by the procedure-like sentence :-

$$R_k \leftarrow R_{k+1}, \ldots, R_n$$

Invoking this sentence in response to the activated call, exactly as in top-down resolution but permitting unification to apply more generally to matchable formulas instead of just to matchable literals, the call R_k in the goal is replaced by a conjunction of calls which then have duplicate occurrences in the new goal; these can clearly be deleted to leave simply :-

$$G_{r+1} : \leftarrow R_1, \ldots, R_{k-1}, R_{k+1}, \ldots, R_n$$

Clearly this is just tantamount to deleting the call R_k from the given goal. Note that this simplification determines that G_r and G_{r+1} are logically equivalent.

Useful call deletions often depend upon suitable instantiation of the goal variables, as shown in the following derivation :-

← *equiv(x,y)*	
← *subset(x,y)*, *subset(y,x)*	[using *equiv* specification]
← *subset(x,x)*, *subset(x,x)*	[instantiation *y:=x*]
← *subset(x,x)*	[deleting first implied call]

from which is inferred the derived procedure :-

equiv(x,x) ← *subset(x,x)*

Deletion of a Valid Call

Any call which is a valid formula of FOPL can be deleted from the current goal to leave a new logically equivalent goal. In practice this kind of simplification will only be applied without explicit justification when the call's validity is trivially provable in FOPL. Deletion of a valid call is, of course, just a special case of the deletion of an implied call as above, since a valid call is necessarily implied by all calls. Below is an example which uses a suitable

instantiation in order to generate a valid call :-

$\leftarrow equiv(x,y)$

$\leftarrow (\forall u)(\ u \in x \leftrightarrow u \in y\ )$ [using *equiv* specification]

$\leftarrow (\forall u)(\ u \in x \leftrightarrow u \in x)$ [instantiation $y:=x$]

$\square$ [deletion of valid call]

from which we conclude the procedure :-

$equiv(x,x) \leftarrow$

Distribution of Connectives

Distribution of the connectives in a call is an especially important means of simplifying the current goal. It is almost always applied with the aim of replacing a moderately complicated call by a conjunction of individually simpler ones. On many occasions this is achieved by distributing $\leftarrow$ through conjunctions or disjunctions, and so a few typical cases of this are enumerated below.

Case 1 : An activated call $(A \leftarrow B_1, \ldots, B_m)$ may be replaced by any non-empty conjunction of calls in which each call is some $(A \leftarrow B_k)$ where $1 \leq k \leq m$. The new goal is logically implied by - but not generally logically equivalent to - the given goal. However, logical equivalence is obtained when the substituted conjunction contains every conjunct $(A \leftarrow B_k)$ such that $1 \leq k \leq m$.

This simplification investigates the problem of showing that a set of assumptions implies A by just attempting to show that some of them individually imply A.

Case 2 : An activated call $(A_1, \ldots, A_m \leftarrow B)$ may be replaced by the conjunction of calls $(A_1 \leftarrow B), \ldots, (A_m \leftarrow B)$. The derived goal is logically equivalent to the given goal. There is no special problem-solving significance in this simplification; it just re-expresses the goal in a more discrete form.

Case 3 : An activated call $(A \leftarrow B_1 \vee \ldots \vee B_m)$ may be replaced by the conjunction of calls $(A \leftarrow B_1), \ldots, (A \leftarrow B_m)$. The derived goal is logically equivalent to the given goal. Like the case above, the goal is merely given a more discrete representation.

Case 4 : An activated call $(A_1 \vee \ldots \vee A_m \leftarrow B)$ may be replaced by any non-empty conjunction of calls in which each call is some $(A_k \leftarrow B)$ where $1 \leqslant k \leqslant m$. The new goal is logically implied by - but not generally equivalent to - the given goal. However, logical equivalence is obtained when the substituted conjunction contains every conjunct $(A_k \leftarrow B)$ such that $1 \leqslant k \leqslant m$.

This simplification investigates the problem of showing that *B* implies a disjunction of alternatives by showing that *B* implies some of them.

These distributions of conditionals may be easily combined to give rules for distributing equivalence connectives in activated calls. Four particularly common cases are shown below, in each of which the derived goal is logically implied by - but not generally equivalent to - the given goal.

Case 5 : An activated call $(A \leftrightarrow B_1, \ldots, B_m)$ may be replaced by the conjunction $(A \leftrightarrow B_1), \ldots, (A \leftrightarrow B_m)$.

Case 6 : An activated call $(A \leftrightarrow B_1 \vee \ldots \vee B_m)$ may be replaced by the conjunction $(A \leftrightarrow B_1), \ldots, (A \leftrightarrow B_m)$.

Case 7 : An activated call $(A_1, \ldots, A_m \leftrightarrow B_1, \ldots, B_m)$ may be replaced by the conjunction $(A_1 \leftrightarrow B_1), \ldots, (A_m \leftrightarrow B_m)$.

Case 8 : An activated call $(A_1 \vee \ldots \vee A_m \leftrightarrow B_1 \vee \ldots \vee B_m)$ may be replaced by the conjunction $(A_1 \leftrightarrow B_1), \ldots, (A_m \leftrightarrow B_m)$.

The above selection of cases is by no means a complete summary of all goal simplifications which proceed by distributing connectives, but is presented just to indicate their style. The frequent need to distribute $\leftarrow$ and $\leftrightarrow$ arises fundamentally from the liberal use made of these connectives by non-clausal specifications. All those goal simplifications which distribute connectives can be reformulated as combined applications of more elementary propositional rules such as De Morgan's laws and the laws of associativity, distributivity and tautology.

Some justification should be given here of the description of these goal transformations as 'simplifications' in spite of the fact that their application usually introduces conjunctions of calls which are syntactically more cumbersome than the calls which they replace. In applying one of these rules to replace a call g by a conjunction $g_1, \ldots, g_n$, the resulting derived goal G_{r+1} is simpler than the given goal G_r in two possible senses. Firstly, the task of solving g is reduced to the separate tasks of solving $g_1, \ldots$ and g_n, each of which is simpler than g because (generally) it only deals with some proper subset of the various atoms occurring in g; normally the solution of g is made tactically simpler by pursuing solutions of its subproblems even though they may not be independent of one another. Secondly, it is often the case that $g_1, \ldots, g_n$ implies g somewhat trivially yet is not equivalent to it, and therefore has less information to deal with; the new calls represent a special way of solving g which is more trivial than the most general way of solving it. For example, solution of the problem :-

$$\leftarrow (u \varepsilon a \leftarrow u \varepsilon b_1)$$

trivially solves :-

$$\leftarrow (u \varepsilon a \leftarrow u \varepsilon b_1, u \varepsilon b_2)$$

and is the case dealt with above as *Case 1*. Problem reduction and problem trivialization are therefore distinct ways of simplifying the solution of goals, and both are typical results of distributing connectives as in the rules above.

Calls which are prefixed by quantifiers may also be replaced by conjunctions of simpler ones by distributing connectives. In particular, suppose that Qg is a call in the current goal G_r of interest, where Q is any prefix of universal quantifiers and g is any formula. Then each of the eight simplification rules above which replace a call g by some conjunction $g_1, \ldots, g_n$ can be generalized to a rule which permits the call Qg to be replaced by the conjunction $Qg_1, \ldots, Qg_2$. This is so by virtue of the validity of the formula :-

$$((\forall x) g(x) \leftarrow (\forall x) g_1(x), \ldots, (\forall x) g_n(x)) \leftarrow (\forall x) (g(x) \leftarrow g_1(x), \ldots, g_n(x))$$

The following example shows such a generalization of both *Cases 3* and *4* above :-

$\leftarrow (\forall u)(\ u\varepsilon y_1 \vee u\varepsilon y_2 \leftarrow u\varepsilon x_1 \vee u\varepsilon x_2)$

$\leftarrow (\forall u)(\ u\varepsilon y_1 \vee u\varepsilon y_2 \leftarrow u\varepsilon x_1),\ (\forall u)(\ u\varepsilon y_1 \vee u\varepsilon y_2 \leftarrow u\varepsilon x_2)$ *[Case 3]*

$\leftarrow (\forall u)(\ u\varepsilon y_1 \leftarrow u\varepsilon x_1),\ (\forall u)(\ u\varepsilon y_2 \leftarrow u\varepsilon x_1),$
$(\forall u)(\ u\varepsilon y_1 \leftarrow u\varepsilon x_2),\ (\forall u)(\ u\varepsilon y_2 \leftarrow u\varepsilon x_2)$ *[Case 4 twice]*

This reduction seeks to show $(x_1 \cup x_2) \subseteq (y_1 \cup y_2)$ by solving each of the more trivial subproblems $x_1 \subseteq y_1$, $x_1 \subseteq y_2$, $x_2 \subseteq y_1$ *and* $x_2 \subseteq y_2$.

Distribution of Quantifiers

Quite often it is useful to distribute quantifiers through connectives. In the simple cases illustrated below, Q is a prefix consisting just of universal quantifiers; this is the most usual circumstance of interest.

Case 1: An activated call $Q(A_1, \ldots, A_m)$ can be replaced by the conjunction $QA_1, \ldots, QA_m$. This just re-expresses the given goal in an equivalent but more discrete form.

Case 2: An activated call $Q(A_1 \vee \ldots \vee A_m)$ can be replaced by any conjunction of calls each with the form QA_k where $1 \leq k \leq m$. The derived goal is logically implied by - but is not generally equivalent to - the given goal. The intuition here is simply that a disjunction can be satisfied by satisfying some of its disjuncts; the quantification here is inconsequential.

Case 3: An activated call $Q(A \leftarrow B_1, \ldots, B_m)$ can be dealt with by *Case 2* by expressing it in disjunctive form, i.e. $Q(A \vee \sim B_1 \vee \ldots \vee \sim B_m)$.

These substitutions can be justifiably regarded as goal simplifications for the same reasons as those advanced in the discussion of connective distributions. Distributions of connectives and quantifiers are often used cooperatively as shown in the following example :-

$\leftarrow (\forall u)(\ (u\varepsilon x \leftarrow u\varepsilon y_1,\ u\varepsilon y_2),\ (u\varepsilon y_1,\ u\varepsilon y_2 \leftarrow u\varepsilon x))$

$\leftarrow (\forall u)(\ u\varepsilon x \leftarrow u\varepsilon y_1,\ u\varepsilon y_2),\ (\forall u)(u\varepsilon y_1,\ u\varepsilon y_2 \leftarrow u\varepsilon x)$

$\leftarrow (\forall u)(\ u\varepsilon x \leftarrow u\varepsilon y_1),\ (\forall u)(u\varepsilon y_1 \leftarrow u\varepsilon x),\ (\forall u)(u\varepsilon y_2 \leftarrow u\varepsilon x)$

where the first step distributes $\forall$ through conjunction whilst the second distributes $\leftarrow$ through conjunction.

Deletion of Quantifiers

Some rather minor goal simplifications proceed by the deletion of quantifiers. The most trivial of these, of course, is the case where a prefix is deleted because the formula which it quantifies has no free occurrences of the prefix variables; there would be no good reason for deliberately arranging such a pre-condition, but the latter could come about instead as a side-effect of other more significant goal transformations.

Another trivial deletion is that of a wholly existential prefix Q_E from a call $Q_E g$ to leave just g; this derives a logically equivalent goal. An example is shown below in which Q_E is $(\exists z)$:-

$$\leftarrow (\exists z \forall u)(\ u \varepsilon y \leftrightarrow u \varepsilon x \vee u \varepsilon z\ )$$
$$\leftarrow \quad (\forall u)(\ u \varepsilon y \leftrightarrow u \varepsilon x \vee u \varepsilon z\ )$$

There is no problem-solving intuition involved here; the rule just exploits Kowalski's goal notation which allows the suppression of explicit existential quantifiers associated with the goal's free variables.

Goals may often be simplified in their quantification by exploiting special properties of elementary relations. A simple example is the exploitation of the 1:1 property of =. Consider the call :-

$$(\forall u)(\ u \varepsilon x \leftarrow u = w\ )$$

assumed to be the activated call in the current goal. Now all instances of u which falsify $u=w$ make the conditional $(\ u \varepsilon x \leftarrow u = w\ )$ vacuously true. Assuming that = is 1:1, the only other instance of u is w, and this instance makes the conditional true if and only if it makes $u \varepsilon x$ true. This means that the call can be replaced by simply $w \varepsilon x$, leaving a goal which is equivalent to the given goal in a theory which has the properties of = as axioms. Strictly this simplification depends upon the implicit axiomatization of = in S and is therefore strictly a goal substitution in the sense defined earlier; but identity is such a fundamental relation, having no specificity with respect to particular problem formulations, that for practical purposes we can treat it as though it were independent of S. An example is shown below in which both the 1:1 property and the symmetry of = are summoned to simplify the goal :-

$\leftarrow (\forall u)(\ u{=}v_1 \vee u\varepsilon x_1 \leftrightarrow u{=}v_2 \vee u\varepsilon x_2\ )$

$\leftarrow (\forall u)((u{=}v_1 \leftrightarrow u{=}v_2),\ (u\varepsilon x_1 \leftrightarrow u\varepsilon x_2))$

$\leftarrow (\forall u)(\ u{=}v_1 \leftrightarrow u{=}v_2),\ (\forall u)(\ u\varepsilon x_1 \leftrightarrow u\varepsilon x_2)$

$\leftarrow v_1{=}v_2,\ v_2{=}v_1,\ (\forall u)(\ u\varepsilon x_1 \leftrightarrow u\varepsilon x_2)$

$\leftarrow v_1{=}v_2,\ (\forall u)(\ u\varepsilon x_1 \leftrightarrow u\varepsilon x_2)$

The inferences used here for the four steps are, respectively, a *Case 8* distribution of $\leftrightarrow$, distribution of $\forall$ through conjunction, a double exploitation of the 1:1 property of = , and finally use of the symmetry of =. The first two steps are goal simplifications proper, and the second two are strictly goal substitutions invoking properties of = from the specification set but more conveniently interpreted as goal simplifications.

5.3 : GOAL SUBSTITUTIONS

In contrast to the kind of simplification rules presented in the last section, which do not enrich the knowledge in the current goal but merely rearrange it or trivialize it, the object of each rule now to be discussed is to derive a new goal by combining the knowledge in the current goal with that expressed in the specification set. It was suggested in Chapter 4 that this process could be viewed as quasi-computation analogous to conventional logic program execution by interpreting the specification set as a procedure set and by interpreting its interactions with derived goals as execution mechanisms. In particular, these mechanisms exhibit features such as call activation (independent or cooperative), procedure invocation (deterministic or non-deterministic) and data transmission (input or output, by argument instantiation). The notion of a *successful* derivation in this context differs slightly from that associated with successful logic program execution in that it is terminated when all the calls in the current goal (if any) are atomic; note that this still allows the possibility of a refutation derivation - that is, when no calls remain to be processed. Observe, then, that whereas successful program execution computes a *solution [*refutation + induced bindings to output variables*]*, a successful procedure derivation generates a *new way [*Horn clause goal + induced bindings to output variables*]* of investigating the original goal.

The inference rules which allow the implementation of the above ideas are, for the most part, a little more elaborate than resolution because of their need to cater for non-atomic calls and procedure headings. As with the previous discussion of rules for goal simplification, the following description of goal substitutions is confined to deal with just those rules which have proved consistently useful in the examples investigated. Fortunately they are few in number, simple in concept and compact in presentation; however, a set of rules proved to be complete would probably not share those attributes.

Inference Rules for Goal Substitution

Each incremental step in the top-down execution of a Horn clause goal consists of selecting one of its conjuncts (that is, activating a call), selecting a resolving input clause (that is, invoking a procedure from the program body) and finally substituting the matched procedure body for the call to the derived goal. The resolution which accomplishes this essentially combines the inference rule *modus tollens* with the unification of literals. This kind of inference has a simple but useful analogue in the process of procedure derivation which only requires an extension of the notion of unification so as to apply to arbitrary formulas rather than just literals. In addition to this simple analogue of program execution, there exist some variants of it which allow slightly more elaborate ways of invoking knowledge in S in order to make a substitution for the activated call. Each of the substitution rules is now discussed in turn.

1] Modus Tollens

It will be helpful to begin with a concrete example; therefore consider the following goal assumed to be the current goal in some procedure derivation :-

$$\leftarrow perm(x',y'),(\forall uv)(u<v \leftarrow consec(u,v,y')),append^*(w,y',y),select(w,x',x)$$

and suppose that the second (non-atomic) call is activated. Assume also that the following sentence occurs in the specification set :-

$$(\forall uv)(u<v \leftarrow consec(u,v,z)) \leftarrow ord(z)$$

and is invoked in response to the activated call. Viewed as a procedure, this sentence has the procedure heading :-

$$(\forall uv)(u<v \leftarrow consec(u,v,z))$$

and the procedure body $ord(z)$. Now the activated call can clearly be matched with the invoked heading by the unifier $\{y':=z\}$. Applying this instantiation together with *modus tollens* then produces the derived goal :-

$$\leftarrow perm(x',z),\ ord(z),\ append^*(w,z,y),\ select(w,x',x)$$

by the substitution of an atomic body for a non-atomic call. As a second example, suppose that the current goal is :-

$$\leftarrow perm(x,y),\ ord(y)$$

and that the invoked procedure from the specification set is :-

$$ord(z) \leftarrow (\forall uv)(u<v \leftarrow consec(u,v,z))$$

Then the unifier $\{y:=z\}$ with *modus tollens* substitutes the procedure body for the call to give :-

$$\leftarrow perm(x,z),\ (\forall uv)(u<v \leftarrow consec(u,v,z))$$

This time a non-atomic body replaces an atomic call.

The inferences just illustrated can be formalized in the following way. Let F be the activated call in the current goal G_r and assume that S either contains or implies a sentence $(F^* \leftarrow F')$ such that F and F^* are unifiable by some θ. The derived goal G_{r+1} is then obtained by substituting F' for F in G_r and applying θ to the result. [Note that to avoid confusion in the naming of variables, this and all other goal substitution rules proceed on the assumption that prior to the inference step the variables in the goal have been named so as to be distinct from those in the invoked sentence, or at least to the extent that no ambiguities arise in the composition of the unifier θ].

A special case of this rule is where S simply implies the assertion $F^* \leftarrow$ which corresponds to F' being merely *true*; in this case the assertion immediately solves the activated call and so effectively just deletes it from the current goal. Clearly these inferences are direct analogues of Horn clause procedure invocation. Observe in particular that the application of θ to the substituted goal acts as a device for transmitting data between calls exactly as in logic program execution. Finally, note that this rule determines that the derived goal is S-implied by - but not generally S-equivalent to - the given goal.

2] Transitivity of Implication

Because the calls in the goal are not limited to atomic formulas, the possibility exists of more elaborate ways in which S may contribute to the substitution of a goal subformula F by some F' originating from S ; indeed most non-trivial procedure derivations depend crucially upon this possibility. The next rule to be presented here deals with a call which is itself a conditional of the form $(F \leftarrow A)$ where both F and A are arbitrary. This call seeks to show that A implies F. Ignoring trivial ways of showing this (for instance, by relying upon the specification set to imply $(F \leftarrow A)$), the simplest general way of solving the call is to show that A implies some F' which itself implies F. Imagine then that S contains or implies the procedure $F^* \leftarrow F'$ where F^* and F are unifiable by some θ. Then the derived goal is obtained by substituting F' for F in the activated call and applying θ to the result. In an analogous way, if the activated call takes the form $(A \leftarrow F)$ and S implies the procedure $F' \leftarrow F^*$ with F^* and F unifiable by some θ, then the derived goal is obtained by substituting F' for F in the call and applying θ to the result. The applicability of these rules is clearly dependent upon the context of F in the goal; that is, unless F is either the consequent or the antecedent of a call which is a conditional formula, these rules will not apply. Observe also that the derived goal is S-implied by - but not generally S-equivalent to - the given goal.

3] S-Equivalence Substitution

The rules presented so far have only considered circumstances in which S implies a conditional sentence. A more powerful kind of goal substitution is possible when S implies an equivalence. Suppose now that S implies a sentence $F^* \leftrightarrow F'$ and that the current goal contains a subformula F which unifies with F^* using some θ. Then a derived goal is obtained by substituting F' for F and applying θ to the result. This rule is more powerful than the others above insofar as it applies independently of the context of F in the given goal; that is, F need not be a call or have any special contextual position in a call. In fact F can even be a whole conjunction of calls, as though all were being simultaneously activated, but normally we shall prefer to apply

the rule in cases where F is a subformula of a particular call, and then consider that call to be the activated one in the current goal. By contrast with Horn clause resolution, observe that derivations in FOPL using rules of this kind allow inferences to be made about the inner structure of calls. Because this particular rule allows F to have an arbitrary context, it has the important property that the derived goal is necessarily S-equivalent to the given goal. It can be regarded as just a variant of the general replacement axiom for deductive logic. At this stage in the discussion of goal substitutions it will be useful to see a concrete example.

Suppose that the current goal seeks to show that the intersection of some pair of sets is a subset of some other set :-

$$\leftarrow (\forall u)\,(\, u \varepsilon y \leftarrow u \varepsilon x_1,\ u \varepsilon x_2\,)$$

and that the specification set contains the set membership axiom below applicable to sets represented by terms :-

$$w \varepsilon v{:}z \leftrightarrow w{=}v \vee w \varepsilon z$$

Now this axiom can be used to express the membership of any of the sets named in the goal in terms of the membership of its constituents. Assume that the set x_1 is chosen as the one to have its membership expressed in more detail in consequence of what S says about the constructibility of sets. Then the following identifications are made to admit an application of the S-equivalence substitution rule :-

$$\begin{aligned} F^* &= w \varepsilon v{:}z \\ F' &= w{=}v \vee w \varepsilon z \\ F &= u \varepsilon x_1 \\ \theta &= \{u{:=}w,\ x_1{:=}v{:}z\} \quad \text{to match } F \text{ with } F^* \end{aligned}$$

so that the derived goal becomes :-

$$\leftarrow (\forall w)\,(\, w \varepsilon y \leftarrow (w{=}v \vee w \varepsilon z),\ w \varepsilon x_2\,)$$

and the single binding to a goal output variable, namely $x_1{:=}v{:}z$, is assigned to the binding environment of the derivation. Note also that the same result could be obtained using one of the *transitivity of implication* rules described in *2]* above by exploiting a weaker fact from S which is implied by the equivalence just used; this fact is just the conditional :-

$$w \varepsilon v{:}z \leftarrow (w{=}v \vee w \varepsilon z)$$

Now the given goal can, just for the sake of example here, be re-written in the form :-

$$\leftarrow (\forall u)(\ u\varepsilon y \vee \sim u\varepsilon x_2 \leftarrow u\varepsilon x_1\ )$$

and then considered to have its activated call in the form $(A \leftarrow F)$, whilst S implies a sentence of the form $F' \leftarrow F^*$ such that F and F^* are unified by $\{u:=w,\ x_1:=v:z\}$. *Transitivity of implication* then derives the new goal :-

$$\leftarrow (\forall w)(\ w\varepsilon y \vee \sim w\varepsilon x_2 \leftarrow w=v \vee w\varepsilon z)$$

which, after re-writing again to eliminate the explicit $\sim$, assumes the same form as the goal just derived by *S-equivalence substitution*.

4] S-Conditional-Equivalence Substitution

The inference rule to be explained now is probably the most frequently used of all the goal substitution rules. Like the one above it has the object of substituting a formula F' for an arbitrary subformula F in the activated call of the goal. Now, however, this substitution is conditional upon some other arbitrary formula F'', and so a call to F'' will appear in the derived goal. The specification set is assumed to imply a conditional equivalence of the form $(F^* \leftrightarrow F') \leftarrow F''$ such that the selected goal subformula F unifies with F^* using θ. The derived goal is obtained by substituting F' for F, applying θ to the result and finally appending the new call F''. The rule allows the context of F in the given goal to be arbitrary, and has the property that the derived goal is S-implied by - but not generally S-equivalent to - the given goal.

A simple example of the rule is shown below which seeks to show that some y is a lower bound for some set x :-

$$\leftarrow (\forall u)(\ y \leqslant u \leftarrow u\varepsilon x\ )$$

The specification set is assumed to contain the following axiom about the constructibility of sets using set union :-

$$(\forall w)(\ w\varepsilon z \leftrightarrow w=v \vee w\varepsilon z'\ ) \leftarrow union^*(v,z',z)$$

Now make the identifications :-

$$F = u\varepsilon x \qquad F' = (w=v \vee w\varepsilon z')$$
$$F^* = w\varepsilon z \qquad F'' = union^*(v,z',z)$$
$$\theta = \{u:=w,\ x:=z\}$$

and apply the rule as just described to give the derived goal :-

$\leftarrow (\forall w)(\ y \leq w \leftarrow w=v \vee w \in z'\ ),\ union^*(v,z',z)$

It should be stated here that the applicability of this rule does have a slight contextual dependence upon the structure of the current goal G_r, although this is not especially associated with the context of its replaced subformula F; rather it is a constraint upon the quantification of the goal variables. A sufficient condition for the rule to be applicable as described earlier is that the free variable set of F'' and the bound variable set of $G'_r\theta$ should be disjoint, where G'_r is the result of substituting F' for F in the current goal G_r, and θ is the unifier of F with F^*. In the example above this condition is satisfied because the free variable set of F'' is $\{v,z',z\}$ whilst the bound variable set of $G'_r\theta$ is $\{w\}$, these being clearly disjoint. Cases where this condition upon the quantification of the goal variables is not satisfied and thus obstructs some substitution for F conditional upon F'' have not occurred during investigation of any of the examples presented in this thesis, and so would not seem to represent a significant limitation in practice.

A final observation about this inference rule is that when F'' is trivialized to just *true*, the *S-conditional-equivalence substitution* reduces to the *S-equivalence substitution*; since the free variable set of F'' is then empty it must be disjoint with the bound variable set of the substituted goal, and so the quantification constraint vanishes. This is consistent with the fact that there are no quantification constraints upon the applicability of the *S-equivalence substitution*.

5] Conditional Transitivity of Implication

This rule could be regarded as a hybrid formed from the two previous rules *2]* and *4]*. There are two variants of it, both of which exploit the transitivity of implication. In the first variant, the activated call has the form $(F \leftarrow A)$ and S implies a sentence of the form $(F^* \leftarrow F') \leftarrow F''$ with F and F^* unifiable by some θ. The derived goal is obtained by substituting F' for F, applying θ and finally appending a new call F''. In the second variant, the activated call has the form $(A \leftarrow F)$ and S implies a sentence of the form $(F' \leftarrow F^*) \leftarrow F''$ with F and F^* unifiable by θ. The derived goal is obtained by substituting F' for F, applying θ to the result and finally appending the new call F''. In each case the context of F in the goal

is specific, and in each case the derived goal is *S*-implied by - but not generally *S*-equivalent to - the given goal. A concrete example of the rule's application is not given here, but an interesting instance of it will be found in the next section dealing with the derivation of recursive procedures.

Summary

The detailed syntactical description of these rules appears a little intimidating, and so it is useful now to give an informal summary of their objectives and preconditions. Given a current goal, the objective of all the rules is to replace some subformula by a formula from the specification set; this will be motivated by the belief that the goal becomes, in some sense, more informed about the problem of interest. In general we would ideally like to replace the selected subformula by an equivalent one, so that the derived goal would certainly not say anything less than did the given goal. Also we would like to choose the subformula without worrying about its context in the goal. The realization of this ideal is the *S-equivalence substitution*, and its precondition is simply that the appropriate equivalence should be implied by the specification set.

The next best approximation to the latter rule is the *S-conditional equivalence substitution* which still allows *F* to be any subformula but no longer preserves *S*-equivalence between the given and the derived goals. The remaining rules just deal with the simplest of the special cases of *F*'s context in the given goal : when it is the consequent or antecedent of a conditional call, or when it is itself a call. Jointly these rules suffice for a great variety of procedure derivations, as will be seen in due course. Their general usefulness can be made apparent by considering the style in which specification sets are typically assembled. For suppose that *R* is the relation for which procedures are to be derived, and is specified in *S* by the sentence :-

$$R \leftrightarrow D_1, \ldots, D_m$$

It may be possible to foresee that those procedures will usefully interrogate some other relations *R'*, *R"*, ... specified in *S* by analogous sentences :-

$$R' \leftrightarrow D'_1, \ldots, D'_{m'},$$

$$R'' \leftrightarrow D''_1, \ldots, D''_{m''}$$

$$\vdots$$

where the conjuncts of the various definiens' are generally non-atomic. It is because they are non-atomic that the derivation of procedures for R from these sentences cannot be accomplished using just *modus tollens*, which poses the need for the other inference rules which exploit deeper relationships between the definiens' obtaining through the sharing of unifiable subformulas. Moreover, it is observed empirically that it is very usual for an individual conjunct, say D''_k, to have the structure $(F^* \leftrightarrow F')$; that is, the specification set implies $(F^* \leftrightarrow F') \leftarrow R''$ by virtue of the 'only-if' half of the R'' definition. This is just the right pre-condition for making an *S-conditional equivalence substitution* of F' for some subformula F in the goal using θ to unify F with F^*. This inference step will therefore introduce a call to R'', and thereby contribute towards the transformation of the goal to a conjunction of entirely atomic calls; this is because R'' is an atom by assumption. *This is a highly typical feature of non-trivial procedure derivation.*

Combining Simplification and Substitution

In general, the strategy which underlies the procedure derivations is disposed towards the substitution of atomic calls for non-atomic calls. A goal simplification tends to increase the goal's amenability to such substitutions by so modifying its syntax that its subformulas can be matched with other formulas in the specification set. Typically a simplification is applied to a call which is not unifiable with any formula in that set; usually its result is to replace that call by a conjunction of simpler ones of which some may be so unifiable.

The contribution of goal substitutions to the pursuit of clausal form is more complex and does not seem capable of some single generalized interpretation. Sometimes it may replace complete non-atomic calls by atoms or vice versa; or it may only replace a highly localized subformula, leaving the rest of the call intact. Yet the most consistent feature of logic procedure derivation is the need to interleave simplifications and substitutions cooperatively. This comes about through the choice of a rather finely discretized approach to the derivation methodology, allowing alternative choices of simplification and substitution at each inference step.

This section closes with two examples of derivations which interleave the rules described here.

Specification Set :

$$\begin{aligned} min(u,x) &\leftrightarrow u\varepsilon x,\ lowerbound(u,x) \\ lowerbound(u,x) &\leftrightarrow (\forall v)(\ u\leqslant v \leftarrow v\varepsilon x\ ) \\ union^*(w,x',x) &\leftrightarrow (\forall v)(\ v\varepsilon x \leftrightarrow v\varepsilon x' \vee v=w\ ) \\ empty(x) &\leftrightarrow (\forall v)(\ v\varepsilon x \leftrightarrow false\ ) \end{aligned}$$

Initial Goal : $\leftarrow min(u,x)$

Derivation :

$\leftarrow min(u,x)$

$\leftarrow u\varepsilon x,\ lowerbound(u,x)$ [modus tollens]

$\leftarrow u\varepsilon x,\ (\forall v)(\ u\leqslant v \leftarrow v\varepsilon x\ )$ [modus tollens]

$\leftarrow (u\varepsilon x' \vee u=w),\ (\forall v)(\ u\leqslant v \leftarrow v\varepsilon x' \vee v=w),\ union^*(w,x',x)$

[S-cond.-equiv.-substitution]

$\leftarrow u=w,\ (\forall v)(\ u\leqslant v \leftarrow v\varepsilon x'\ ),\ (\forall v)(\ u\leqslant v \leftarrow v=w),\ union^*(w,x',x)$

[2 simplifications]

$\leftarrow u=w,\ (\forall v)(\ u\leqslant v \leftarrow false),\ u\leqslant w,\ union^*(w,x',x),\ empty(x')$

[S-cond.-equiv.-substitution]

[1:1 of = to simplify]

$\leftarrow u=w,\ u\leqslant w,\ union^*(w,x',x),\ empty(x')$

[delete valid 2nd call]

$\leftarrow union^*(w,x',x),\ empty(x')$ [u:=w to solve first two calls]

at which point the calls are all atomic; the inferred procedure is :-

$$min(w.x) \leftarrow union^*(w,x',x),\ empty(x')$$

which just gives a useful basis procedure for certain recursive procedures for *min*.

Specification Set :

$$consec(u,v,z) \leftrightarrow (\exists i)(item(u,i,z),\ item(v,i+1,z))$$
$$first(z,u) \leftrightarrow item(u,1,z)$$
$$rest(z,z') \leftrightarrow (\forall ui)(item(u,i,z) \leftrightarrow item(u,i-1,z') \vee (i=1, first(z,u)))$$

Initial Goal : $\leftarrow consec(u,v,z)$

Derivation :

$\leftarrow consec(u,v,z)$

$\leftarrow (\exists i)(item(u,i,z),\ item(v,i+1,z))$ *[modus tollens]*

$\leftarrow item(u,i,z),\ item(v,i+1,z)$ *[deletion of quantifier]*

$\leftarrow item(u,i,z),\ item(v,i,z'),\ rest(z,z')$

[conditional transitivity of implication by virtue of S implying :-

$$(\forall ui)(item(u,i,z) \leftarrow item(u,i-1,z')) \leftarrow rest(z,z')$$

and also employing some primitive arithmetic]

$\leftarrow first(z,u),\ first(z',v),\ rest(z',z)$

[modus tollens twice, inducing the binding i:=1]

at which point all calls are atomic, giving the derived procedure :-

$$consec(u,v,z) \leftarrow first(z,u),\ first(z',v),\ rest(z,z')$$

This is just the non-recursive procedure for *consec* which seeks a consecutive pair in a list by specifically inspecting the first pair.

5.4 : SOME TECHNIQUES FOR PROCEDURE DERIVATION

Derivation of Recursive Procedures

Most interesting computations are either recursive or iterative. Recursive computations emanating from logic program execution are obtained from recursive procedures in the program; iterative behaviour may arise either by bottom-up invocation of recursive procedures, or by top-down invocation of recursive procedures using a stack-overwriting mechanism or, more trivially, by the action of the interpreter in performing incremental search. Examples of these possibilities were given in Chapter 3. In this section we examine the ways in which recursive procedures are typically derived from non-recursive specifications.

A simple example of such a derivation begins with a specification of the w^{th} Fibonacci number due to De Moivre; here the predicate *fib(u,w)* holds when *u* is this number :-

$$fib(u,w) \leftrightarrow u = \frac{1}{\sqrt{5}}(\Phi^w - \hat{\Phi}^w)$$

The constant symbol Φ abbreviates $\frac{1}{2}(1+\sqrt{5})$ and $\hat{\Phi}$ abbreviates $\frac{1}{2}(1-\sqrt{5})$. These constants are related by two sentences which are included in the specification set together with the *fib* definition above :-

$$\Phi^{-1} + \Phi^{-2} = 1 \qquad \hat{\Phi}^{-1} + \hat{\Phi}^{-2} = 1$$

It is also assumed that *S* implicitly contains some simple axioms of arithmetic dealing with addition, multiplication and exponentiation; these could easily be made explicit by introducing predicates *plus*, *times* and *exp*, but for conciseness in what follows it is more convenient to use function symbols to construct arithmetic expressions and then perform 'quasi-arithmetic' upon them.

The objective of the derivation below is to derive a recursive procedure for *fib* which can then be used in a program for computing Fibonacci numbers.

$$\leftarrow fib(u,w)$$

$$\leftarrow u = \frac{1}{\sqrt{5}} * (\Phi^w - \hat{\Phi}^w)$$

$$\leftarrow u = \frac{1}{\sqrt{5}} * (\Phi^w * 1 - \hat{\Phi}^w * 1)$$

$\leftarrow u = \frac{1}{\sqrt{5}} * (\Phi^w * (\Phi^{-1} + \Phi^{-2}) - \hat{\Phi}^w * (\hat{\Phi}^{-1} + \hat{\Phi}^{-2}))$ *[arithmetic]*

$\leftarrow u = \frac{1}{\sqrt{5}} * (\Phi^{w-1} - \hat{\Phi}^{w-1} + \Phi^{w-2} - \hat{\Phi}^{w-2})$ *[arithmetic]*

$\leftarrow u = \frac{1}{\sqrt{5}} * (\Phi^{w-1} - \hat{\Phi}^{w-1}) + \frac{1}{\sqrt{5}} * (\Phi^{w-2} - \hat{\Phi}^{w-2})$ *[arithmetic]*

$\leftarrow u = u_1+u_2,\ fib(u_1,w-1),\ fib(u_2,w-2)$ *[modus tollens twice]*

The derived procedure is then just the familiar Fibonacci identity :-

$$F_w = F_{w-1} + F_{w-2}$$

Kowalski's paper (51) discusses the computational properties of programs which use this procedure. It should be noted that the inferences used in the above derivation are just *modus tollens* and so require only the simplest kind of subformula substitution. The recursiveness of *fib* is inherited here from that of the exponential relation which is implicitly assumed in the specification :-

$$x^{i+j} = x^i * x^j$$

and which in predicate form would be written as the recursive procedure for *exp* :-

$$exp(x,k,y) \leftarrow plus(i,j,k),\ exp(x,i,y_1),\ exp(x,j,y_2),\ times(y_1,y_2,y)$$

Before considering the underlying philosophy of derivations such as this, two more examples will be shown now whose productions of recursive procedures appear to have different origins from that above. Consider then the derivation of the recursive *go** procedure for which Kowalski gives the specification :-

$$go^*(x,z) \leftrightarrow (\ go(z) \leftarrow go(x)\ )$$

A very simple derivation now proceeds as follows :-

$\leftarrow go^*(x,z)$

$\leftarrow (\ go(z) \leftarrow go(x)\ )$ *[modus tollens]*

$\leftarrow (\ go(z) \leftarrow go(y)\ ),\ go^*(x,y)$ *[cond. trans. of implication]*

$\leftarrow go^*(y,z),\ go^*(x,y)$ *[modus tollens]*

so that the procedure inferred is :-

$$go^*(x,z) \leftarrow go^*(x,y),\ go^*(y,z)$$

It may be helpful to clarify the second step by observing that the call in the second goal takes the form $(A \leftarrow F)$ whilst the specification set implies a sentence $(F' \leftarrow F^*) \leftarrow F''$, namely :-

$$(go(\hat{z}) \leftarrow go(\hat{x})) \leftarrow go^*(\hat{x},\hat{z})$$

The *conditional transitivity of implication* rule then admits a substitution for $go(x)$ conditional upon the introduced call $go^*(x,y)$ as follows :-

match F and F^* with unifier $\theta = \{x:=\hat{x}\}$;

substitute F' for F , and apply θ; then add F'' giving :-

$$\leftarrow (go(z) \leftarrow go(\hat{z}))\theta,\ go^*(\hat{x},\hat{z})\ ;$$

then make the renaming $\{\hat{z}:=y,\ \hat{x}:=x\}$ to give :-

$$\leftarrow (go(z) \leftarrow go(y)),\ go^*(x,y)$$

It should be observed that the derived procedure is recursive by virtue of declaring the transitivity of go^*; this recursiveness is not inherited from that of any other problem-specific relations as was the case in the Fibonacci example. Assertions of other general properties of relations such as reflexivity and associativity are also necessarily recursive and may often be useful for computational purposes; for instance, the conventional recursive procedure for *append* is just a specialization of the general property of associativity for the appending operation.

The third example given here illustrates perhaps the most common way of deriving recursive procedures. This proceeds by decomposing selected sub-projections of the relations of interest. To demonstrate this, suppose that the given goal has the subformula $v \varepsilon x$ and hence investigates in some way the question of whether an element v can be found in a set x. Now consider the projection of ε associated with its first argument position and, in particular, that sub-projection $\{v \mid v \varepsilon x\}$ of it determined by some choice of x. This sub-projection can be expressed arbitrarily in terms of its component members and subsets. For instance, if we have some algorithmic intuition about the way the membership of x should be investigated during execution, we might favour the decomposition $x = x_1 \cup x_2 \cup \{v'\}$; this relationship between the components of the sub-projection above is conveniently summarized by the predicate $union^{**}(x_1,x_2,v',x)$.

which is specified in FOPL by :-

$$union^{**}(x_1,x_2,v',x) \leftrightarrow (\forall v)(\ v\varepsilon x \leftrightarrow v\varepsilon x_1 \vee v\varepsilon x_2 \vee v=v'\ )$$

and expresses $\{v \mid v\varepsilon x\} = \{v \mid v\varepsilon x_1\} \cup \{v \mid v\varepsilon x_2\} \cup \{v \mid v=v'\}$.

Note that the above sentence trivially implies a conditional equivalence of the form $(F^* \leftrightarrow F') \leftarrow union^{**}(x_1,x_2,v',x)$, which just states what substitution can be made for a predicate $v\varepsilon x$ subject to the condition that the sub-projection $\{v \mid v\varepsilon x\}$ is decomposed as described by $union^{**}(x_1,x_2,v',x)$. To observe the usefulness of this we revisit the *lowerbound* relation and pursue the derivation of a recursive procedure for it as follows :-

$\leftarrow$ *lowerbound*(u,x)

$\leftarrow (\forall v)(u\leqslant v \leftarrow v\varepsilon x)$ *[modus tollens, invoking assumed lowerbound spec^n.]*

$\leftarrow (\forall v)(u\leqslant v \leftarrow v\varepsilon x_1 \vee v\varepsilon x_2 \vee v=v')$, $union^{**}(x_1,x_2,v',x)$

[S-cond.-equiv.-substitution for subformula $v\varepsilon x$]

$\leftarrow (\forall v)(u\leqslant v \leftarrow v\varepsilon x_1)$, $(\forall v)(u\leqslant v \leftarrow v\varepsilon x_2)$, $(\forall v)(u\leqslant v \leftarrow v=v')$, $union^{**}(x_1,x_2,v',x)$

[distributing $\leftarrow$ through $\vee$ and then distributing $\forall$ through conjunction]

$\leftarrow$ *lowerbound*(u,x_1), *lowerbound*(u,x_2), $u\leqslant v'$, $union^{**}(x_1,x_2,v',x)$

[modus tollens twice, for first two calls, and 1:1 of = to simplify the third call]

This clearly gives a recursive Horn clause procedure. An informal explanation of the intuition underlying the derivation is as follows : for given u and x, solving the call *lowerbound*(u,x) requires the examination of all members in the ε sub-projection $\{v \mid v\varepsilon x\}$; but subject to the condition $union^{**}(x_1,x_2,v',x)$ this may be accomplished by three individual examinations of all instances of v in, respectively, $\{v \mid v\varepsilon x_1\}$, $\{v \mid v\varepsilon x_2\}$ and $\{v \mid v=v'\}$; on inspection of these examinations it is seen that each one investigates the question of whether u is a lowerbound for some set, which is finally expressed in terms of the atomic predicates afforded by the original formulation. It is especially important to note the role of the *S-conditional-equivalence substitution* in this technique for obtaining recursive procedures through the decomposition of sub-projections.

The same technique can be employed for relations of any -arity. For instance, to show that a list x is ordered requires examination of all pairs (u,v) satisfying $consec(u,v,x)$. The sub-projection $\{(u,v) \mid consec(u,v,x)\}$ of the first two argument positions of *consec* can be decomposed conveniently as the union :-

$$\{(u,v) \mid consec(u,v,x')\} \cup \{(u',v) \mid first(x',v)\}$$

which is expressed by the conditional sentence :-

$$(\forall uv)(consec(u,v,x) \leftrightarrow consec(u,v,x') \vee (u=u', first(x',v))) \leftarrow append^*(u',x',x)$$

This sentence is a consequence of the specification for $append^*(u',x',x)$ which holds when u' is the first member of x and x' is the rest of x. Similarly the analogous sentence in the previous example was just a consequence of the specification of the *union*** relation. [However, note that in the present example the sentence is not simply the only-if half of a specification for *append**, as the following counter-example shows : choose $x = (a,b,a,b)$, $x' = (a,b,a)$ and $u' = b$; these instances satisfy the conclusion of the conditional but do not satisfy $append^*(u',x',x)$.] A derivation of a recursive procedure for *ord* now proceeds very easily :-

$\leftarrow ord(x)$

$\leftarrow (\forall uv)(u<v \leftarrow consec(u,v,x))$

$\leftarrow (\forall uv)(u<v \leftarrow consec(u,v,x') \vee (u=u', first(x',v))), append^*(u',x',x)$

$\leftarrow (\forall uv)(u<v \leftarrow consec(u,v,x')), (\forall uv)(u'<v \leftarrow first(x',v)), append^*(u',x',x)$

$\leftarrow ord(x'), u'<v', first(x',v'), append^*(u',x',x)$

[by invoking an axiom that asserts that the <u>first</u> relation is many:1 , viz :-

$(\forall v)(v=v' \leftrightarrow first(x',v)) \leftarrow first(x',v')$

which allows an S-cond.-equiv. substitution for the subformula $first(x',v)$ in the goal's second call; then exploit 1:1 of = to simplify.]

A rather tidier derivation expresses the desired decomposition a little differently using the conditional equivalence :-

$$(\forall uv)(consec(u,v,x) \leftrightarrow consec(u,v,x') \vee (u=u', v=v')) \leftarrow split(x,u',v',x')$$

where the *split* relation is the one used in the example at the end of Section 4.5 ; the derivation of the desired *ord* procedure now follows exactly as shown in that example, resulting in :-

$$ord(x) \leftarrow split(x,u',v',x'),\ u'<v',\ ord(x')$$

Then $split(x,u',v',x')$ can be shown to be implied by the conjunction $(first(x',v'),\ append^*(u',x',x))$ as a later exercise.

In each of the three examples considered, a recursive procedure for some relation R arose through the transformation of an initial goal $\leftarrow R$ into a goal $\leftarrow R_1, \ldots, R_n$ such that at least one R_i turned out to be some substitution instance $R\theta$ of R. This was achieved by a goal-oriented symbolic execution of the conjuncts in the definiens D of R which were introduced into the derivation by invoking R's definition from the specification set. Eventually some R_i became recognizable as $D\theta$ and so was replaced by the S-equivalent atom $R\theta$, thereby introducing recursiveness into the procedure inferred from the derivation. This technique was developed by Burstall and Darlington (10) and also by Manna and Waldinger (61), the former applying it to the derivation of sets of executable recursion equations and the latter applying it to the derivation of LISP programs from specifications. Burstall and Darlington have given the terms 'folding' and 'unfolding' to the acts of definiens-substitution and definiens-replacement which underlie the process described above. Transformations of recursion equations closely resemble transformations of Horn clause procedures, since the two formalisms share many similarities. Clark seems to have been the first to apply seriously the ideas of Burstall and Darlington to the correctness-preserving improvement of Horn clause programs. Application of the same ideas to the derivation of logic procedures from FOPL specifications was investigated soon after by Hogger (38), by Clark and Sickel (15) and - in a less obvious way - by Bibel (3).

As a final note in this section, it should not be thought that the derivation of recursive procedures depends upon the 'fold/unfold' paradigm. Logic has the curious property that a derived procedure

$$R \leftarrow R_1, \ldots, R_n$$

necessarily S-implies

$$R \leftarrow R_1, \ldots, R_n, R\theta$$

for any arbitrary R and θ.

Derivation of Basis Procedures

Suitable bases for recursive procedures may be derived from specifications using the same inference rules as have been shown here for deriving other kinds of logic procedures. In general they arise from trivializations which render calls immediately solvable without requiring further recursive invocations. There are various ways in which calls may be trivially solved. Perhaps the simplest way is to instantiate their variables with known solutions towards which the associated recursive computations are known to converge. An example of this is shown below which derives a basis for the *consec* relation :-

$\leftarrow$ *consec(u,v,z)*
$\leftarrow$ *(∃i)(item(u,i,z), item(v,i+1,z))* *[modus tollens]*
$\leftarrow$ *item(u,i,z), item(v,i+1,z)* *[delete quantifier]*
$\leftarrow$ *item(u,1,z), item(v,2,z)* *[instantiate i:=1]*

and thus infer the basis :-

consec(u,v,z) $\leftarrow$ *item(u,1,z), item(v,2,z)*

This procedure is sufficient to terminate computation initiated by a solvable call to the *consec* procedure set consisting of that procedure together with the recursive procedure :-

consec(u,v,z) $\leftarrow$ *rest(z,z'), consec(u,v,z')*

This is because in the case where the call is solvable the computation must generate a list whose first and second members are respectively the two quoted in the initial call.

The *consec* example above is only concerned with the discovery of any instances of *u* and *v* which satisfy *consec(u,v,z)*. However, in other problems it is necessary to investigate *all* the instances in some sub-projection of the relation of interest. We have seen how this can be represented in derived recursive procedures by exploiting conditional decompositions of such sub-projections. The appropriate bases for computations generated by these means typically deal with the trivial sub-projections that are eventually computed from a succession of decompositions, for instance, sub-projections which are empty or just contain one individual. A simple example of this can be considered in connection with the *lowerbound* relation, beginning with a derivation :-

$\leftarrow$ *lowerbound(u,x)*

$\leftarrow$ *(∀v)(u⩽v ← vϵx)*

Here there is no apparent instantiation of the call's variables which immediately solves the call through the sole agency of goal simplification. Now the usual algorithm employed for the problem of testing whether a given *u* is indeed a lower bound for a given set *x* makes use of a recursive (but iteratively implementable) procedure :-

lowerbound(u,x) ← *union*(v,x',x)*, *u⩽v*, *lowerbound(u,x')*

which selects successive members *v* from *x* and compares them with *u*. The requisite basis deals with the case where there is no such member *v*, signalled by the recursive procedure's failure to solve the call to *union**. Suppose then that a specification is given of the empty set :-

empty(x) $\leftrightarrow$ *(∀v)(vϵx* $\leftrightarrow$ *false)*

Then the derivation above can be continued by invoking this axiom in the context of an *S-conditional-equivalence substitution* as follows :-

$\leftarrow$ *(∀v)(u⩽v ← false)*, *empty(x)*	*[S-cond.-equiv. substitution]*
$\leftarrow$ *empty(x)*	*[deletion of valid 1st call]*

from which the *lowerbound* basis is inferred :-

lowerbound(u,x) ← *empty(x)*

This is a rather round-about way of proving an obvious theorem about *lowerbound*, but it is clearly desirable that the advocated inference rules should cater for the trivial theorems as well as the less trivial ones in order to merit any claim for their general applicability.

Whilst the simple treatment of basis derivation shown above is adequate for many cases, there are nevertheless more subtle ways of providing bases. Kowalski's *go** relation offers such an example, where a particular instantiation allows the deletion of a valid call :-

$\leftarrow$ *go*(x,z)*	
$\leftarrow$ *(go(z)* ← *go(x))*	*[modus tollens]*
$\leftarrow$ *(go(x)* ← *go(x))*	*[instantiate z:=x]*
□	*[delete valid call]*

resulting in the familiar *go** basis :-

go(x,x)* ←

This basis just expresses the general property of symmetry in the *go** relation, and does not arise through consideration of successive decomposition of the relations named in the program body.

A rather more exotic example is provided by the *palin** relation discussed briefly in Section 3.1 . Recall that the predicate *palin*(z',z)* holds when some result *x* of appending *z'* to the reverse of *z* is a palindrome; this is specified easily by :-

$$palin^*(z',z) \leftrightarrow (\exists xz^*)(palin(x) \leftarrow append(z^*,z',x),\ reverse(z,z^*))$$

Now suppose that the motivation for deriving some procedures for *palin** is to find a procedure set which behaves better than that below :-

$$palin(x) \leftarrow append(z^*,z,x),\ reverse(z,z^*)$$
$$palin(x) \leftarrow append(z^*,u.z,x),\ reverse(z,z^*)$$

These could be the procedures trivially implied by a high-level *palin* specification of the meaning of 'palindrome' which avoided reference to the individual indexed members of *x*. In Section 3.1 it was shown that very satisfactory behaviour could be obtained from a *palin** procedure set having one recursive procedure and two bases; here we explain the derivation of those procedures. Consider, then, a derivation for *palin** which assumes that the naive (non-deterministic) *palin* procedures above are available as axioms in the specification set.

$\leftarrow palin^*(z',z)$

$\leftarrow (palin(x) \leftarrow append(z^*,z',x),\ reverse(z,z^*))$ *[modus tollens and delete quantifier]*

Now treat the first *palin* procedure above as an assertion $F^* \leftarrow$ which unifies with the call in the derived goal by $\theta = \{z':=z\}$; thus *modus tollens* will give an immediate refutation. Similarly, invoking the second *palin* procedure as though it were an assertion will also give a refutation with the unifier $\theta' = \{z':=u.z\}$. These two ways of terminating the derivation with the facts available give the two desired bases :-

$$palin^*(z,z) \leftarrow$$
$$palin^*(u.z,z) \leftarrow$$

Discussion of the palin* recursion is a little out of context here, but below is an outline of its derivation. It is only necessary to add to the specification set the further axioms :-

append(z",z',x) ← append(z,u.z',x), append(z*,u.nil,z")*

reverse(u.z,z") ← reverse(z,z), append(z*,u.nil,z")*

of which the first just states a simple consequence of the general associativity of the appending operation, whilst the second is just the familiar recursive *reverse* procedure. These admit a fairly straightforward derivation :-

← palin(u.z',z)*

← (palin(x) ← append(z,u.z',x),reverse(z,z*))* *[as for the bases]*

← (palin(x) ← append(z",z',x), reverse(u.z,z"))

[substituting for each of the inner antecedents by invoking the append and reverse axioms above to exploit transitivity of implication]

← palin(z',u.z)*

and hence produce the recursive procedure :-

palin(u.z',z) ← palin*(z',u.z)*

Finally it may also be noted that the procedure which solves calls to *palin* by solving calls to *palin** can also be derived trivially using the *palin** specification. Noting generally that a sentence *A ↔ (B ← C)* logically implies *B ← C,A* , the *palin** specification similarly implies the non-recursive procedure :-

palin(x) ← append(z,z',x), reverse(z,z*), palin*(z',z)*

Choosing the instantiations *z:=nil* and *z':=x* and invoking trivial properties of *reverse* and *append* then gives :-

palin(x) ← palin(x,nil)*

It is interesting to observe here that although investigation of whether *x* is a palindrome requires examination of all its members - and hence a recursive *palin** procedure to achieve this incrementally - it has not proved necessary to consider as a potential basis the limiting case where *x* is the empty list; in general, the computation apparently terminates before reaching a state where it is involved in processing empty lists generated by successive decomposition. In reality, of course, empty lists *are* inspected, but not through the direct agency of the program's explicit procedures; instead they are

generated by the interpreter's unification procedure which is presented with the task of matching two identical instances of z by a call to either $palin^*(z,z)$ or $palin^*(u.z,z)$, this being performed member by member by the interpreter until only empty lists remain to be matched; this implicit convergence to a basis comparing empty lists is just a consequence of the implicit role of the unification mechanism. A similar example was discussed in Section 3.1 in the case of the procedure set for the list equality relation.

In the course of deriving basis procedures it is frequently necessary to solve several calls conjointly. This can be illustrated in the task of deriving a basis for the program which tests whether x is a palindrome by testing whether x is its own reverse. The appropriate specification set for this purpose is :-

$$palin(x) \leftrightarrow reverse(x,x)$$

$$reverse(x,y) \leftrightarrow (\exists w)(length(x,w), (\forall ui)(item(u,i,x) \leftrightarrow item(u,w+1-i,y)))$$

$$length(x,w) \leftrightarrow (\forall i)(\ 1 \leqslant i \leqslant w \leftrightarrow (\exists u) item(u,i,x))$$

$$empty\text{-}list(x) \leftrightarrow (\forall ui)(item(u,i,x) \leftrightarrow false)$$

which admits an innocuous derivation :-

$\leftarrow palin(x)$

$\leftarrow reverse(x,x)$

$\leftarrow length(x,w),\ (\forall ui)(item(u,i,x) \leftrightarrow item(u,w+1-i,x))$

$\leftarrow (\forall i)(\ 1 \leqslant i \leqslant w \leftrightarrow (\forall u) item(u,i,x)), (\forall ui)(item(u,i,x) \leftrightarrow item(u,w+1-i,x))$

Now these calls can be solved conjointly and trivially by assuming the case where x is the empty list, which makes $item(u,i,x)$ false for all u,i and by also letting w be 0, which makes $1 \leqslant i \leqslant w$ false for all i. Introducing these assumptions has the result of making each of the two substituted calls valid formulas, and adds a call to $empty\text{-}list(x)$ in consequence of the obvious *S-conditional-equivalence substitution*. Therefore the inferred procedure is the expected basis for *palin* :-

$$palin(x) \leftarrow empty\text{-}list(x)$$

A cautionary counter-example is provided by the problem of computing the minimum of a set using the *min* procedure shown earlier. After pursuing the derivation :-

$\leftarrow min(u,x)$

$\leftarrow u \in x,\ (\forall v)(\ u \leqslant v \leftarrow v \in x\ )$

it would not be possible to obtain a basis by attempting to solve the second call using the *empty* specification to substitute *false* for its subformula $v \in x$. The reason for this is that the first call would then be inconsistent with the newly-introduced call *empty(x)* ; the substitution inference could be applied soundly to obtain this result, but would just produce an unsolvable goal. Instead, the appropriate derivation is that shown at the end of Section 5.3, which transforms the goal above to one which seeks the minimum u in a set x' satisfying $x = \{w\} \cup x'$, and then solves this for the trivial case where x' is the empty set; the result of which is the inferred procedure :-

$$min(w,x) \leftarrow union^*(w,x',x),\ empty(x')$$

Completeness of Derived Procedure Sets

We say that a procedure set P is complete for a relation R when it is capable of computing all individuals in R , provided that similarly complete procedure sets are also given for all other relations which might by investigated in P. It is also meaningful to say that a procedure set P is complete for R with respect to a given class of goals; then it must be able to compute all individuals in the sub-projection of R determined by that class of goals.

Provision for ensuring completeness is not explicit in the presentation of logic procedure derivations given here, just as provision for ensuring a search through all ways of solving a goal is not explicit in the text of an executed logic program; we just assume that the search through the derivation graph employs an implicit labelling system which indicates the points at which choices have been made in processing the derivation goal.

Choices of goal transformation which introduce a branch into the tree of derivations making up a synthesis arise in both goal simplification and goal substitution. Some of our simplification rules preserve S-equivalence whilst others do not; if, for instance, we simplify $\leftarrow (A \leftarrow B,C)$ to $\leftarrow (A \leftarrow B)$, then a systematic approach for obtaining a complete procedure set will also require a derivation which

explores the alternative simplified goal $\leftarrow (A \leftarrow C)$. This will lead perhaps to a number of procedures in the derived program which solve the problem at hand in different ways. A different way of organizing their derivation would be to defer goal simplification and instead process the goal first given in some equivalence-preserving way, until its body eventually became transformed into the disjunction of the bodies of all the inferred procedures; but for all but the most trivial problems this is grossly over-cumbersome, and it is then better to concentrate attention on individual derivations in conjunction with proper observance of points where several possibilities need to be explored along different branches in order to ensure completeness.

Alternatives also arise in choice of goal substitutions. When *S*-equivalence substitution is used then no loss of information occurs when generating a new goal. Of course, there might well be alternative *S*-equivalence substitutions which could be applied, but this must not be confused with our immediate problem of dealing with the consequences of loss of *S*-equivalence. *S*-equivalence is usually lost when applying the *S*-conditional-equivalence substitution rule, because the appended condition may represent only one of several ways of replacing the selected goal subformula. Suppose that we use a lemma $(F^* \leftrightarrow F') \leftarrow F_1''$ in order to substitute for a goal subformula. Then it may be, according to the particular problem formulation, that *S* implies the sentence $(F_1'' \vee F_2'')$; in this event the use of the former lemma is associated with just one of two possible branches which must be explored in order for the synthesis to be complete for the resources provided by *S* ; the other branch corresponds to the application of the alternative lemma $(F^* \leftrightarrow F') \leftarrow F_2''$ in order to replace that same goal subformula. A typical example of this is where we derive a complete procedure set for the *subset* relation by replacing the subformula $u \varepsilon x$ in the *subset* definiens by either *false* or $(u=u' \vee u \varepsilon x')$ respectively in two *subset* derivations. The completeness of the two cases has to be justified by the independent assumption that the specification set implies $(F_1'' \vee F_2'')$ where F_1'' is *empty(x)* and F_2'' is $(\exists u'x')union^*(u',x',x)$. Without that assumption, which comes about from a proper data structure characterization of the data structures in this case (but not in all cases, by any means), we could not say that the two inferred procedures would compute all individuals in *subset*. Again, it is too cumbersome to arrange all the equivalence-preserving knowledge to be held in a single goal; we pursue alternative goals for the various cases of substitution, and argue for completeness afterwards.

5.5 : DERIVATION OF DATA-ACCESSING PROCEDURES

A set of derived procedures for some relation will usually refer to some other relations as well. In Section 5.1 it was explained how the successive derivation of procedure sets in the course of deriving a complete program body imposed a hierarchical structure upon the synthesis methodology. Whereas it has been advocated that the higher-level procedures should only refer abstractly to their data structures, the lower-level procedures will refer to data structures using concrete representations. The discussion which now follows deals firstly with the derivation of procedures which access the components of term representations, and then (briefly) with the derivation of procedures which interrogate assertional data structures.

Procedures for Accessing Terms

Illustration of the ways in which procedures may be derived for accessing terms can be accomplished by concentrating upon lists as the data structures of interest. Lists are, of course, the primary data structures manipulated by most programs, and so the restriction of the following discussion to lists is not a very serious one. Throughout the present work, the fundamental notion associated with lists is that of indexed membership as expressed by the predicate *item(u,i,x)*, which holds when *u* is the i^{th} member of list *x*. This predicate is treated as a primitive constructor for specifying other computationally useful relations like *append**, and has the following axioms associated with it :-

$$A1 : \quad (\exists w)\, length(x,w) \leftarrow$$

$$A2 : \quad length(x,w) \leftrightarrow (\forall i)(1 \leqslant i \leqslant w \leftrightarrow (\exists u)\, item(u,i,x))$$

$$A3 : \quad (\forall u)(item(u,i,x) \leftrightarrow u=u' \leftarrow item(u',i,x)$$

The elementary relation $\leqslant$ in *A2* is assumed to have only non-negative integers in its domain. *A1* and *A2* jointly determine that every list *x* has an integral length $w \geqslant 0$ and that every list *x* with length *w* has some member *u* associated with each index $i \mid 1 \leqslant i \leqslant w$. In particular, *A1* and *A2* jointly imply $1 \leqslant i$ if *item(u,i,x)* holds. Axiom *A3* determines that the member associated with any index is unique.

In deriving procedures which manipulate lists, it will normally be assumed that axioms *A1-A3* are included in the specification set.

Now consider the term representation of lists which has been used so frequently in the examples presented previously, that is, terms constructible using . and *nil*. The meaning of indexed membership for such representations of lists is specified by the rather recursive sentence :-

$$item(u,i,z) \leftrightarrow (\exists vz')(z=v.z',(item(u,i-1,z') \vee (u=v,i=1)))$$

This can be rewritten as shown below so as to separate two cases according to whether or not z is empty :

$$item(u,i,nil) \leftrightarrow false$$
$$item(u,i,v.z') \leftrightarrow item(u,i-1,z') \vee (u=v,i=1)$$

where *nil* is just a Skolem function symbol. These considerations establish the preliminaries for derivation of procedures accessing lists represented as terms.

As a first example of the way in which the above knowledge can be applied, suppose that the composition of some program body demands the supply of procedures which simply investigate the question of an element's membership in a list, ignoring the question of its position, if any. The predicate symbol used to express list membership is $\hat{\epsilon}$; the specification of $\hat{\epsilon}$ is as follows :-

$$u\hat{\epsilon}z \leftrightarrow (\exists i)item(u,i,z)$$

This, together with *A1-A3*, is sufficient for the derivation of a procedure set for $\hat{\epsilon}$. This can be shown quite briefly as follows :-

$\leftarrow u\hat{\epsilon}z$
$\leftarrow item(u,i,z)$ *[modus tollens, and delete quantifier]*
$\leftarrow item(u,i-1,z') \vee (u=v,i=1)$ *[modus tollens + induced binding $z:=v.z'$]*

The derivation so far has proceeded virtually deterministically. Now, however, the derivation branches according to whether we have $i=1$ or $i>1$. Axioms *A1* and *A2* determine that these cases are exhaustive. If the first case is assumed, the derivation continues :-

$\leftarrow item(u,0,z') \vee (u=v,1=1)$ *[instantiate $i:=1$]*
$\leftarrow false \vee u=v$ *[because S implies, from A1-A3, the fact :-*

$(item(u,i,z) \leftrightarrow false) \leftarrow i\leqslant 0$]

□ *[simplifying the disjunction to u=v*
and then trivially solving with instantiation v:=u]

Pursuing the alternative case instead :-

← *item(u,i-1,z')* ∨ *(u=v,i=1)* *[returning to the branch point]*
←*(item(u,i-1,z')* ∨ *(u=v,false)), i>1*

[S-cond.-equiv. substitution using :-

(i=1 ↔ false) ← i>1]

← *item(u,i-1,z'), i>1*	*[simplifying]*
← *item(u,i-1,z')*	*[deleting second call S-implied by the first]*
← *item(u,j,z')*	*[instantiating j:=i-1]*
← *(∃j)item(u,j,z')*	*[inserting explicit quantifier]*
← *uêz'*	*[modus tollens]*

The two procedures inferred from these derivations are then just :-

$$u\hat{\in}u.z' \leftarrow$$
$$u\hat{\in}v.z' \leftarrow u\hat{\in}z'$$

Of course, these are just trivial consequences of an alternative recursive $\hat{\in}$ specification which makes no use of the notion of indexed membership :

$$u\hat{\in}z \leftrightarrow (\exists vz')(z=v.z',\ (u\hat{\in}z' \vee u=v))$$

but which is less general than the former specification in that it is specific to a particular representation of lists (by terms). For instance, the former specification immediately provides an $\hat{\in}$ procedure for accessing lists represented by sets of *item* assertions instead :-

$$u\hat{\in}z \leftarrow item(u,i,z)$$

As a general methodological principle it seems desirable to derive accessing procedures using a data structure axiomatization like *A1-A3* even though those procedures might be obvious from the outset; deriving them from a common foundation (like the notion of indexed membership) gives coherence and integrity to the general task of interfacing procedures and their data structures.

The question of membership regarding a particular element is perhaps the easiest question that can be asked about some list. A slightly more elaborate query is that which also requires the position of a list's member. Clearly questions about both membership and position can be investigated by some appropriate call to our *item* predicate provided that a suitable procedure set for *item* is made available. Such a procedure set is trivially implied by the specification given previously for lists represented by terms; the procedures are simply :-

$$item(u,1,u.z') \leftarrow$$
$$item(u,i,v.z') \leftarrow item(u,i-1,z')$$

and their derivation is so trivial that it is not worth formalizing it. Of course, if the list in question was represented by a set of *item* assertions then it would not be necessary to devise any other procedures in order to investigate a call to *item*; this reflects the fact that assertional data structure representations can be regarded as accessing procedures in their own right.

In many cases we need accessing procedures which not only manipulate individual members of lists, but also whole fragments of those lists. A simple case of this is found in any algorithm which searches a list sequentially by a succession of decompositions, each of which inspects and then discards the first member of the current fragment. Typically we would employ the *append** procedure for this task, since a call $append^*(u',z',z)$ can deal with both the first member u' and the rest z' of the list z. Knowledge about *append** can be composed from the elementary properties of *item*, as is now demonstrated.

A simple specification of *append** is as follows :-

$$append^*(u',z',z) \leftrightarrow first(z,u'), rest(z,z')$$

where *first* and *rest* are in turn specified by :-

$$first(z,u') \leftrightarrow item(u',1,z)$$
$$rest(z,z') \leftrightarrow (\forall ui)(item(u,i-1,z') \leftrightarrow item(u,i,z), i>1)$$

These axioms allow us to choose a list representation, define the meaning of *item* for that representation and then derive an appropriate procedure set which accesses the representation in whatever way is desired. Only the simplest case is illustrated here, where lists are represented by the orthodox terms already considered above. This permits the following trivial derivations :-

$\leftarrow$ *first(z,u')*

$\leftarrow$ *item(u',1,z)* *[modus tollens]*

$\square$ *[modus tollens, inducing z:=u'.z']*

giving the accessing procedure for the first member of a list :-

first(u'.z',u') $\leftarrow$

For the *rest* procedure :-

$\leftarrow$ *rest(z,z')*

$\leftarrow$ *(∀ui)(item(u,i-1,z')* $\leftrightarrow$ *item(u,i,z), i>1)* *[modus tollens]*

$\leftarrow$ *(∀ui)(item(u,i-1,z')* $\leftrightarrow$ *(item(u,i-1,z')* $\vee$ *(u=v,i=1)),i>1)*

[S-equiv. substitution, inducing z:=v.z']

$\leftarrow$ *(∀ui)(item(u,i-1,z')* $\leftrightarrow$ *(item(u,i-1,z'),i>1)* $\vee$ *false)*

[distributing conjunction through $\vee$
and then simplifying using properties of >]

$\leftarrow$ *(∀ui)(item(u,i-1,z')* $\leftrightarrow$ *item(u,i-1,z'))*

[deleting i>1 S-implied by item(u,i-1,z')]

$\square$ *[modus tollens]*

giving the accessing procedure for the 'rest' of a list :-

rest(v.z',z') $\leftarrow$

The appropriate procedure for *append** for this list representation is then obtained trivially by combining the two results above :-

$\leftarrow$ *append*(u',z',z)*

$\leftarrow$ *first(z,u'), rest(z,z')* *[modus tollens]*

$\leftarrow$ *rest(u'.ẑ',z')* *[modus tollens, inducing z:=u'.ẑ']*

$\square$ *[modus tollens, inducing z':=ẑ']*

giving the familiar assertion for decomposing a list :-

append(u',z',u'.z')* $\leftarrow$

These derivations thus provide a coherent foundation for those simple theorems about the constituents of terms which we conventionally employ as useful accessing procedures.

The decomposition of lists by calls to *append** is important to many of the logic programs derived in this thesis. To see how such calls can be introduced during procedure derivation, observe that the specifications just given for *append**, *first* and *rest* jointly imply an alternative *append** specification which refers directly to the list's indexed members; the new definiens for *append** is obtained by conjoining the definiens' of *first* and *rest* and slightly simplifying the result, which is :-

$$append^*(u',z',z) \leftrightarrow (\forall ui)(item(u,i,z) \leftrightarrow item(u,i-1,z') \vee (u=u',i=1))$$

When this specification is included amongst the axioms used by some derivation, it can clearly be summoned for the purpose of making an *S-conditional-equivalence substitution* which replaces some goal subformula consisting of just an *item* predicate by an *S-cond.-equivalent* disjunction, and adds a call to *append** to the goal. Thus the goal is modified by an assumption of a particular way of accessing the lists to which it refers.

In order to aid the comprehension of subsequent derivations, it will be instructive now to see how the above process is applied to the task of deriving the conventional recursive *append* procedure :-

$$append(u'.z_1',z_2,u'.z') \leftarrow append(z_1',z_2,z') \cdot$$

which is essentially a generalization of *append** which allows the first argument to be any list rather than just a single member. The *append* relation is specified rigorously by :-

$$append(z_1,z_2,z) \leftrightarrow (\exists j)(length(z_1,j),$$
$$(\forall ui)(item(u,i,z) \leftrightarrow item(u,i,z_1)$$
$$\vee\ item(u,i-j,z_2)))$$

Assuming a specification set which contains this definition of *append* together with that for *append** and the list axioms *A1-A3*, it is now desired to derive a procedure for *append* which accomplishes its task by decomposing the lists in question in the specific manner expressed by *append**. There are two ways of going about this : the 'low-level' way which just instantiates the lists quoted in the *append* specification with their term representations and then simplifies the result to obtain the *append* procedure shown above; and the 'high-level' way which pursues a derivation that introduces explicit calls to *append**, deferring commitment to any particular

choice of list representation until the end of the derivation. An example of the low-level approach to the synthesis of a procedure set for *append* is given in Clark's paper (12), although the style of his approach is rather different from that employed here. The high-level derivation proceeds as follows :-

$\leftarrow$ *append*(z_1,z_2,z)

$\leftarrow$ *length*(z_1,j), $(\forall ui)$(...) *[modus tollens, delete quantifier]*

It is convenient just for presentation's sake to show separate derivations emanating from these two calls, combining them later.

$\leftarrow$ *length*(z_1,j) *[investigating the first call]*

$\leftarrow$ $(\forall i)(1 \leqslant i \leqslant j \leftrightarrow (\exists u)\,item(u,i,z_1))$ *[modus tollens]*

$\leftarrow$ $(\forall i)(1 \leqslant i \leqslant j \leftrightarrow (\exists u)(item(u,i-1,z_1') \vee (u=u',i=1)))$, $append^*(u',z_1',z_1)$

[S-cond.-equiv. substitution to express the decomposition of z_1 *by append*]*

$\leftarrow$ $(\forall i)(1 \leqslant i \leqslant j \leftrightarrow (\exists u)\,item(u,i-1,z_1') \vee (\exists u)(u=u',i=1)), append^*(u',z_1',z_1)$

[distributing $\exists$ *through* $\vee$*]*

$\leftarrow$ $(\forall i)(1 < i \leqslant j \vee i=1 \leftrightarrow (\exists u)\,item(u,i-1,z_1') \vee i=1)$, $append^*(u',z_1',z_1)$

[exploiting simple property of $\leqslant$ *over integers, and simplifying* $(\exists u)(u=u',i=1)$ *to just* $i=1$*]*

$\leftarrow$ $(\forall i)(1 < i \leqslant j \leftrightarrow (\exists u)\,item(u,i-1,z_1'))$, $append^*(u',z_1',z_1)$

[simplifying by cancellation of each disjunct $i=1$*]*

$\leftarrow$ $(\forall \hat{i})(1 \leqslant \hat{i} \leqslant j-1 \leftrightarrow (\exists u)\,item(u,\hat{i},z_1'))$, $append^*(u',z_1',z_1)$

[instantiation $\hat{i}:=i-1$*, and properties of* $\leqslant$*]*

$\leftarrow$ *length*$(z_1',j-1)$ *[modus tollens]*

This derivation shows that when the condition $append^*(u',z_1',z_1)$ is imposed, a call *length*(z_1,j) can be replaced by *length*$(z_1',j-1)$; an obvious result but deserving of proof from the assumed problem formulation. Next we pursue a derivation from the second call of the original goal, again exploiting the condition expressing the list decomposition. This proceeds as follows :-

$$\leftarrow (\forall ui)(item(u,i,z) \leftrightarrow item(u,i,z_1) \vee item(u,i\text{-}j,z_2))$$

$$\vdots$$

$$\leftarrow (\forall ui)(item(u,i,z') \leftrightarrow item(u,i,z'_1) \vee item(u,i\text{-}j,z_2)),$$
$$append^*(u',z',z),\ append^*(u',z'_1,z_1)$$

[after simplifying the result of making two obvious S-cond.-equiv. substitutions using append]*

If the two derivations are now combined, a single application of *modus tollens* produces the desired procedure :-

$$append(z_1,z_2,z) \leftarrow append^*(u',z'_1,z_1),\ append(z'_1,z_2,z'),$$
$$append^*(u',z',z)$$

If the decision is made now to use the orthodox term representation, some trivial macroprocessing of the *append** calls gives the familiar procedure :-

$$append(u'.z'_1,z_2,u'.z') \leftarrow append(z'_1,z_2,z')$$

It is important to appreciate the way in which comparatively high-level accessing procedures like *append** can be assimilated into goals which investigate decomposable data structures. The derivation above offers us the freedom at the end to pick an alternative representation upon which to implement the calls to *append**, perhaps to improve run-time efficiency. In order to justify these remarks, an example will be given in the later discussion of access to assertional data structures, in which two calls to *append** appearing in a context similar to that above will be implemented firstly upon a non-orthodox term representation, followed by an interesting transformation of the result which leads to their efficient implementation upon a representation using a set of assertions instead.

Derivations like those above may of course be undertaken for data structures other than lists. Access to representations of sets is also quite a common requirement. There, it is usual to employ the constructors : and $\emptyset$ with accessing procedures like ε, *union** and *union*. Although sets are mathematically simpler than lists, the customary term representation of sets is similar to that of lists where . and *nil* are used. A curious consequence of this is that sets of accessing procedures which manipulate terms representing sets tend to be more complicated than their analogues which treat the terms

as representations of lists. Suppose, for instance, that some procedure contains a call *append*(u,z',z)*. Then we know that if *z* is decomposable then its term representation will have the form *v.y* ; moreover, the call computes the unique solution *u:=v, z':=y* by a single invocation of a trivial procedure :-

append(u,y,u.y)* ←

By contrast, suppose that some procedure contains a call *union*(u,x',x)* where *x* is some term *v:y* representing a decomposable set. Then there are generally many instances of *u* and *x'* representing solutions of the call for that particular *x*. Additionally, however, even for a particular instance of *u* chosen from *x*, there will exist generally many solutions of *x'*. For example, if *x* is *a:b:c:∅* then two possible solutions are (*u:=a, x':=b:c:∅*) and (*u:=a, x':=c:b:∅*). A procedure set for *union** capable of computing all representations of the various solutions is as follows :-

union(u,y,u:y)* ←
union(u,v:y',v:y)* ← *union*(u,y',y)*
union(u,y',y)* ← *union*(v,y",y)*,*union*(u,y',v:y")*

Whether or not all these are necessary depends upon the context in which they are summoned to access data. The following procedure set for the *subset* relation :-

subset(∅,y) ←
subset(x,y) ← *union*(u,x',x)*, *u∈y*, *subset(x',y)*

only requires one of the *union** procedures to solve the problem of showing that the set *a:b:c:∅* is a subset of the set *c:b:a:∅*, namely :-

union(u,x',u:x')* ←

But to solve the same problem using the program :-

← *subset(a:b:c:∅,c:b:a:∅)*

subset(∅,y) ←
subset(x,y) ← *union*(u,x',x)*, *union*(u,y',y)*, *subset(x',y')*

additionally requires the second *union** procedure. All three *union** procedures are needed to solve the problem of showing those two sets to be equivalent using the procedure set :-

equiv(∅,∅) ←
equiv(x,y) ← *union*(u,x',x)*, *union*(u,x',y)*

Procedures for Accessing Assertions

The merits of assertional data structures were remarked upon in Section 3.2 . There it was argued that when terms are used to represent data structures it is often not possible to construct procedures allowing direct access to their components. This consideration encourages the search for alternative representations such as sets of assertions which allow efficient access and storage. When some derived procedure contains a call to some procedure P, it may be convenient to arrange for this call to be investigated by interrogating a set of assertions of the form $P \leftarrow$ which provide immediate solutions to the call. Whether or not this is sensible depends upon the circumstances of the overall problem. Consider, for example, the problem of discovering whether or not a set $S_1=\{b,c\}$ is a subset of $S_2=\{a,b,c,d\}$ using the procedures :-

$$subset(x,y) \leftarrow empty(x)$$
$$subset(x,y) \leftarrow union^*(u,x',x),\ u\varepsilon y,\ subset(x',y)$$

Here it might seem reasonable to represent the set S_2 by the assertions :-

$$a\varepsilon S_2 \leftarrow \quad b\varepsilon S_2 \leftarrow \quad c\varepsilon S_2 \leftarrow \quad d\varepsilon S_2 \leftarrow$$

thus providing immediate solutions to any call $u\varepsilon S_2$. On the other hand it seems much less reasonable to provide a set of assertions :-

$$union^*(b,S_1',S_1) \leftarrow \quad union^*(c,S_1'',S_1') \leftarrow \quad empty(S_1'') \leftarrow$$

to represent S_1. Nevertheless the more obvious treatment of S_1 :-

$$b\varepsilon S_1 \leftarrow \quad c\varepsilon S_1 \leftarrow$$

cannot be manipulated in any straightforward way by a call to $union^*(u,x',S_1)$, since this call must generate a representation of x'. Section 3.2 examined a similar case concerning a palindrome-testing program which inspected a list x represented by a set of assertions. It was shown that, in principle, a computation could be obtained by giving the name $mid(x)$ to the middle of x and then providing procedures capable of computing the indexed members of $mid(x)$. But that kind of program really requires mixed top-down and bottom-up invocations to be practical, together with a sensible strategy for interleaving them. A more satisfactory approach to that problem was shown which used alternative procedures containing explicit pointers to the components of the assertional data structure representation. This

approach can also be followed for dealing with problems like the one above for *subset*. Before demonstrating this for the *subset* problem, it will be instructive to firstly examine in detail an analogous problem which processes lists rather than sets; this is the problem of showing that a given list is ordered.

Suppose that the following naive procedure set has already been derived for the *ord* relation :-

ord(x) ← length(x,0)
ord(x) ← length(x,1)
ord(x) ← append(u',x',x), append*(v',x",x'), u'<v', ord(x')*

using the specification set :-

A1-A3 [the list axioms]
ord(x) ↔ (∀uvi)(u<v ← item(u,i,x), item(v,i+1,x))
append(u',x',x) ↔ (∀ui)(item(u,i,x)↔item(u,i-1,x') ∨ (u=u',i=1))*

It will be assumed here that any list *L* of interest, such as *(3,5,7,9)* will be represented by a set of assertions :-

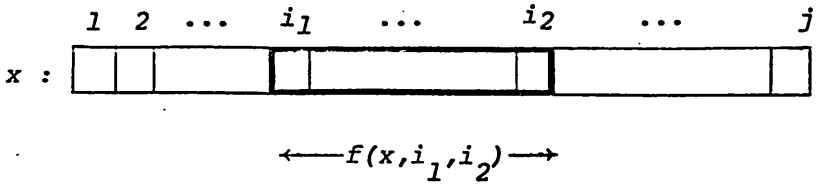
item(3,1,L) ← item(7,3,L) ← length(L,4) ←
item(5,2,L) ← item(9,4,L) ←

so that the orderedness of lists with length < 2 can be investigated by the *ord* bases by direct interrogation of the data structure's *length* component. Thus the central problem remaining is how to deal with the general case (length ⩾ 2) where the orderedness of *L* has to be ascertained by successive inspection of its consecutive pairs. This poses the problem of implementing the calls to *append** in the *ord* procedure responsible for this successive decomposition upon the given list representation.

Consider a fragment of any list *x* which extends from its i_1^{th} member up to and including the i_2^{th} member, assuming $i_1 \leqslant i_2$. By analogy with the naming *mid(x)* in the palindrome problem, here we can name this fragment as $f(x,i_1,i_2)$ where *f* is just a Skolem function symbol signifying the existence of such a fragment. The indexed membership in the fragment is then specified logically by an *S-equivalence* *F* ↔ F'* , namely :-

$$item(u,i-i_1+1,f(x,i_1,i_2)) \leftrightarrow item(u,i,x),\ i_1 \leqslant i < i_2$$

It will probably be helpful to portray the relationship between x and $f(x,i_1,i_2)$ as follows :-



This sentence specifying the meaning of *item* for fragments of x represented that way is assumed also to be included in the specification set. The primary objective now is to derive procedures for *append** capable of accessing the components of such fragments; in particular, such procedures would then be capable of accessing x represented as the fragment $f(x,1,j)$ where j is the length of x.

Consider then a derivation from the first call in the recursive *ord* procedure above, renaming (just for convenience) the last two arguments :-

$\leftarrow$ *append**(u',y',y)

$\leftarrow$ *append**$(u',f(x,i_1+1,i_2),\ f(x,i_1,i_2))$ *[an intuitive choice of fragments of x, inducing* $y:=f(x,i_1,i_2)$ $y':=f(x,i_1+1,i_2)$ *]*

$\leftarrow (\forall ui)(item(u,i,f(x,i_1,i_2)) \leftrightarrow item(u,i-1,f(x,i_1+1,i_2)) \vee (u=u',i=1))$ *[modus tollens]*

$\leftarrow (\forall ui)(item(u,i+i_1-1,x),(i_1 \leqslant i+i_1-1 \leqslant i_2) \leftrightarrow (item(u,i+i_1-1,x),(i_1 \leqslant i+i_1-1 \leqslant i_2)) \vee (u=u',i=1))$

[S-equiv. substitution for each of the item predicates and some arithmetic on the indices (trivial)]

$\leftarrow (\forall ui)(item(u,i+i_1-1,x),(i_1 \leqslant i+i_1-1 \leqslant i_2) \leftarrow u=u',i=1)$

[a goal simplification which solves $\leftarrow(A \leftrightarrow A \vee B)$ *by solving* $\leftarrow(A \leftarrow B)$ *]*

$\leftarrow i_1 \leqslant i_2,\ item(u',i_1,x)$ *[using 1:1 of = to simplify]*

What this derivation has shown is that a call :-

$$\leftarrow append^*(u', f(x,i_1+1,i_2),\ f(x,i_1,i_2))$$

which decomposes the list $f(x,i_1,i_2)$ can be solved by showing that $i_1 \leqslant i_2$ (essentially checking that the term is a well-formed list) and then confirming that u' is the i_1^{th} member of x. Similarly it is easy to pursue a directly analogous derivation from the second *append** call in the recursive *ord* procedure which shows that to solve a call :-

$$\leftarrow append^*(v', f(x,i_1+2,i_2),\ f(x,i_1+1,i_2))$$

it suffices to show $i_1+1 \leqslant i_2$ and confirm that v' is the $(i_1+1)^{\text{th}}$ member of x. This derivation induces the bindings :-

$$y' := f(x,i_1+1,i_2)$$
$$y'' := f(x,i_1+2,i_2)$$

when initiated by the goal $\leftarrow append^*(v',y'',y')$. Each derivation allows us to infer a non-recursive procedure for *append** which can be used to macroprocess out the associated call to *append** in the recursive *ord* procedure, the result of which is :-

$$ord(f(x,i_1,i_2)) \leftarrow i_1+1 \leqslant i_2,\ item(u',i_1,x),\ item(v',i_1+1,x),\ u'<v',\ ord(f(x,i_1+1,i_2))$$

Now the list whose orderedness is in question is simply x, and so it is pertinent to consider under what circumstances x and $f(x,i_1,i_2)$ name the same list. By exploiting the *item* specification together with the *length* specification, it is trivial to establish a sufficient condition for this in the form of an *S-conditional-equivalence* :-

$$(\forall ui)(item(u,i,x) \leftrightarrow item(u,i,f(x,i_1,i_2))) \leftarrow i_1=1,\ length(x,i_2)$$

Then this sentence and the *ord* specification trivially imply yet another sentence in that form :-

$$(ord(x) \leftrightarrow ord(f(x,1,i_2))) \leftarrow length(x,i_2)$$

We shall use this result to transform the derived *append**-free recursive *ord* procedure into one which dispenses with the explicit references to fragments represented by terms.

It can be shown that, for computational purposes, the Skolem

symbol f is superfluous in the argument structure of a call to *ord*. An easy way to eliminate references to f is to introduce a new predicate *ord** specified by :-

$$ord^*(x,i_1,i_2) \leftrightarrow ord(f(x,i_1,i_2))$$

This and the previous sentence jointly imply a procedure which investigates orderedness by making a call to *ord** :-

$$ord(x) \leftarrow length(x,j),\ ord^*(x,1,j)$$

Moreover, the *ord** specification admits a trivial transformation of the recursive *ord* procedure which eliminates its references to f, the result of which is :-

$$ord^*(x,i_1,i_2) \leftarrow i_1{+}1 \leqslant i_2,\ item(u',i_1,x),\ item(v',i_1{+}1,x),\ u'<v',\ ord^*(x,i_1{+}1,i_2)$$

Some further inferences easily show that a suitable basis for this is simply :-

$$ord^*(x,j,j) \leftarrow$$

which deals with the case of a fragment of x consisting just of a unit list.

The consequence of these transformations of the original procedure for *ord* is that orderedness can now be investigated using the new procedure set :-

$$ord(x) \leftarrow length(x,0)$$
$$ord(x) \leftarrow length(x,j),\ ord^*(x,1,j)$$
$$ord^*(x,j,j) \leftarrow$$
$$ord^*(x,i_1,i_2) \leftarrow i_1{+}1 \leqslant i_2,\ item(u',i_1,x),\ item(v',i_1{+}1,x),\ u'<v',\ ord^*(x,i_1{+}1,i_2)$$

This procedure set gives excellent behaviour when executed in conjunction with a set of assertions representing the list of interest. It behaves very much as a conventional program which maintains a loop index i_1 varying from 1 up to j to select the consecutive pairs from x. Note that the only list named explicitly in the new procedure set is x. All other lists which underlie the logic of the algorithm (that is, the various fragments considered in the above derivations) are represented only implicitly by the pointers in the last two argument positions of the *ord** procedures. It is these explicit pointers which enable direct access to members and hence efficient behaviour.

The general approach taken to the above example is also applicable to the *subset* problem with which we opened the discussion of access to assertional data structure representations. All that is necessary is to arrange that members of the set S_1 are labelled in some way which enables explicit pointers to select them systematically . Additionally it is necessary to assert the cardinality of S_1. Putting these arrangements into effect for the example considered earlier, the sets' representations would be :-

$member(b,1,S_1) \leftarrow$ $\quad a\varepsilon S_2 \leftarrow$ $\quad c\varepsilon S_2 \leftarrow$

$member(c,2,S_1) \leftarrow$ $\quad b\varepsilon S_2 \leftarrow$ $\quad d\varepsilon S_2 \leftarrow$

$cardin(S_1,2) \leftarrow$

and the transformed *subset* procedure set would be :-

$subset(x,y) \leftarrow cardin(x,0)$

$subset(x,y) \leftarrow cardin(x,j),\ subset^{\dagger}(x,1,j,y)$

$subset^{\dagger}(x,j,j,y) \leftarrow$

$subset^{\dagger}(x,i_1,i_2,y) \leftarrow i_1+1 \leqslant i_2,\ member(u,i_1,x),$
$u\varepsilon y,\ subset^{\dagger}(x,i_1+1,i_2,y)$

This programming style just makes explicit the iterative search through a set of assertions. Its implementation can be made very efficient for an interpreter already equipped to access sets of procedures for normal computational purposes. The most attractive feature of such programs in contrast to their counterparts for accessing terms is that they incur almost no burden upon the unifying mechanism, and do not generate a complex binding environment in order to represent the explicit products of data structure decompositions. Thus, whilst the derivation of accessing procedures for terms is of theoretical interest, and can be an interesting first stage in a synthesis, the derivation of accessing procedures for assertions is of greater practical importance.

CHAPTER 6

EXAMPLES OF PROGRAM DERIVATIONS

PREVIEW

Six examples are given in this chapter in order to illustrate the application of logic procedure derivation as described in Chapter 5. All of them consider very simple computational problems which are nevertheless sufficient to demonstrate a variety of interesting programming styles and derivations.

The first example considers the familiar problem of list reversal, initially deriving the standard recursive algorithm which employs a binary appending operation for representing both the input and output lists. The first salient point of the reversal example is encountered in that exercise, namely the need for the preliminary derivation of useful lemmas. Here we wish to pursue an intuitively obvious goal substitution, but the initial specification set does not contain an axiom immediately suitable for that purpose. It is shown how the inference rules normally applied to derivation goals can also be used to derive the desired lemma and so allow the required goal substitution. It is next shown how the general reversal program can be specialized in a number of ways to give iterative algorithms, although none of these are as satisfactory as the *reverse** program (introduced in Chapter 3), which is now derived here. It is seen that the *reverse** program is obtained by exploiting the procedures already produced for the recursive program, together with invocation of an associative property of the appending operation.

The second example is the problem of searching a given list for

duplicates. Whereas the list reversal algorithms shared more or less comparable efficiency, three algorithms of differing efficiencies are given here for the duplicate problem. The last of these is especially interesting in its specification, employing a special argument in the logic which acts as a stack recording the distinct members found during the search. The derivation of the latter algorithm makes use of the procedures from the more naive algorithms to construct an important lemma from which a useful goal substitution is procured. The use of case analysis for goal transformation is particularly high-lighted here by the natural occurrence of disjunctive calls in the goal and the way in which those calls are simplified and regrouped.

The third example deals with the generation of factorial tables, presenting a number of algorithms with various efficiencies. The last one is particularly interesting in the structure of its specification, which encodes in the logic an assumption about the context in which the procedures of an earlier algorithm are invoked; the newly derived algorithm is then able to avoid a contextual check (in fact, a test to ensure a correct run-time goal structure) which the previous algorithm could not.

The fourth example is that of comparing the frontiers of two binary trees. Here the key to the desired algorithm lies in the relatively low-level matters concerning the data structures. A lemma describing the associativity of tree construction is proved and then used to provide an essential equivalence substitution. Also, it is shown that the most obvious basis can be usefully generalized in order that the comparison of frontiers may, at some stage during the execution, be accomplished immediately by a single unification rather than by continuing sequential comparisons under the agency of the program's recursive decomposition procedures.

The fifth example is a particular summation problem in a matrix. An interesting use of Kowalski's quasi-bottom-up programming style is shown which, instead of its normal role of just reversing the direction of a top-down computation, here generates an efficient quasi-parallel bottom-up summation in contrast to the most obvious top-down solution of the problem in which distinct sums are computed in sequence.

The sixth and last example is just the eight queens problem, which is included just to emphasize how the powerful control mechanisms of logic interpreters allow a far more pleasing representation of the problem than is possible in conventional programming languages.

6.1 : PROGRAMS FOR LIST REVERSAL

Specifying the Problem

The simple problem of reversing a list almost invariably appears in examples of new methods for developing programs, and has already received some attention in earlier parts of the present work. It is appropriate, then, to examine the reversal problem in detail to see how well it can be dealt with in the logic procedure derivation methodology. The relation of chief interest initially is the 2-ary relation *reverse* which can be specified in terms of the basic list constructor *item* as follows :-

$$reverse(x,y) \leftrightarrow (\exists z)(length(x,z),$$
$$(\forall ui)(item(u,i,x) \leftrightarrow item(u,z+1-i,y)))$$

It is also assumed that the specification set used in formulating this problem contains the list axioms *A1-A3* established in the last chapter, together with general properties of elementary relations like = and $\leqslant$. Specifications of other relations will also be introduced to the set when their need becomes apparent.

It is perhaps worth noting that there is a little redundancy in the above definition of *reverse* in that the predicate *length(x,z)* can be deleted from the definiens to leave an *S-equivalent* sentence; that is, in order that x and y shall be specified as reverse to one another, it is only necessary to insist that there shall exist some z such that u is in the i^{th} position of x if and only if it is also in the $(z+1-i)^{th}$ position of y. This is because axioms *A1-A3* constrain the *item* relation in such a way as to ensure that any instance of z satisfying this last requirement must be the length of both x and y. However, rather than just permitting this to be enforced implicitly, it is more satisfactory to include the fact that z must be the length of one of the lists in the *reverse* definition itself; this helps us to keep the special status of z explicit within the derivations.

The Recursive Reversal Program

The first *reverse* synthesis examined here pursues the well-known recursive algorithm which arbitrarily splits the given list into two parts, reverses them, and finally composes the results into the reverse of the given list. This is just one of the most simple ways of

arranging for the systematic decomposition of the input list, which must necessarily underlie the logic of a reversal algorithm. Here we consider just the most conceptually simple way of splitting that list, namely that of generating two lists x_1 and x_2 satisfying the property that x results from appending x_2 to x_1. This idea prompts the introduction of the *append* specification :-

$$append(x_1,x_2,x) \leftrightarrow (\exists z_1)(length(x_1,z_1), (\forall ui)(item(u,i,x) \leftrightarrow item(u,i,x_1) \vee item(u,i-z_1,x_2)))$$

which is now added to the specification set. Note here that the predicate $length(x_1,z_1)$ must be included in the definiens for *append*, notwithstanding the presence of axioms *A1-A3*. This completes the initial knowledge necessary to derive the recursive reversal algorithm. It is now desired to derive the high-level procedures for *reverse* by beginning :-

$\leftarrow reverse(x,y)$

$\leftarrow length(x,z)\ ,\ (\forall ui)(item(u,i,x) \leftrightarrow item(u,z+1-i,y))$

If the assumption that x is to be split in the manner described using a call to *append* is now to be incorporated into the derivation, it would be pleasing to accomplish this by invoking the sentence specifying *append* with the object of making an appropriate goal substitution. Intuitively we can foresee a goal substitution which replaces the predicate $item(u,i,x)$ by another formula referring to the members of x_1 and x_2, conditional upon some appended call to *append*; in other words, a normal *S-conditional-equivalence substitution.*

To achieve the objective just outlined, it is necessary to have a sentence implied by S with the form :-

$$(F^* \leftrightarrow F') \leftarrow append(x_1,x_2,x)$$

It might be thought at first sight that the only-if half of the *append* definition would provide such a sentence, but unfortunately this turns out not to be so due to the obstructive presence of the existential quantifier. This is not to say that there is no such sentence implied by S; there is indeed such a sentence, but it will have to be derived as a lemma before the above derivation can be developed in the desired fashion. The need for such preliminaries just reflects the fact that, in general, assembling a naive set of axioms in S does

not ensure that those axioms can be immediately applied to the derivation goal using our most-favoured inference rules. Lemma generation, as well as goal transformation, plays an important role in logic program development, and is not often easy to organize in an obviously top-down style. The need for a lemma may be induced by top-down reasoning from the current goal, considering how a goal substitution might be procured; but the deduction of the lemma itself may be quite a bottom-up process.

In the present example, an appropriate lemma can be obtained by showing that the length of a list is unique, as expressed by :-

$$(length(x,z') \leftrightarrow z=z') \leftarrow length(x,z)$$

and then using this sentence to make an *S-conditional-equivalence substitution* in the definition of *append*. For ease of presentation, suppose that we had the sentences :-

$$A(x) \leftrightarrow (\exists u)(C(u,x),\ B(u,x)) \quad [like\ the\ append\ def^{n}.]$$
$$(C(u,x) \leftrightarrow u=v\ ) \leftarrow C(v,x) \quad [like\ uniqueness\ of\ length]$$

Then these would imply :-

$$\vdash (A(x) \leftrightarrow (\exists u)(u=v, B(u,x))) \leftarrow C(v,x)$$
$$\vdash (A(x) \leftrightarrow B(v,x)) \leftarrow C(v,x)$$
$$\vdash B(v,x) \leftarrow A(x),\ C(v,x)$$

By analogy, in the present case we would thereby have shown that *S* implied the *S-conditional-equivalence* :-

$$(\forall ui)(item(u,i,x) \leftrightarrow item(u,i,x_1) \vee item(u,i\text{-}z_1,x_2)) \leftarrow append(x_1,x_2,x),\ length(x_1,z_1)$$

Proof of the lemma that states that the length of a list is unique can be obtained by using axiom *A2* as a means of making a subformula substitution into its own definiens as follows :-

$$S \vdash length(x,z) \leftrightarrow (\forall i)(1 \leqslant i \leqslant z \leftrightarrow (\exists u)item(u,i,x)) \quad [axiom\ A2]$$
$$\vdash length(x,z') \leftrightarrow (\forall i)(1 \leqslant i \leqslant z' \leftrightarrow (\exists u)item(u,i,x)) \quad [just\ renaming]$$
$$\vdash (length(x,z') \leftrightarrow (\forall i)(1 \leqslant i \leqslant z' \leftrightarrow 1 \leqslant i \leqslant z)) \leftarrow length(x,z)$$

[making an S-cond.-equiv. substitution]

$$\vdash (length(x,z') \leftrightarrow z=z') \leftarrow length(x,z) \quad [exploiting\ properties\ of\ \leqslant\ to\ simplify]$$

With this somewhat digressive exercise accomplished, the pre-condition for a goal substitution which expresses the decomposition of x by a call to *append* is now established. Now it has also been anticipated that the output list y is constructed by an appending operation as well; therefore a substitution can be made for each *item* predicate in the second call to give :-

$$\leftarrow length(x,z),\ (\forall ui)(item(u,i,x_1) \vee item(u,i-z_1,x_2) \leftrightarrow item(u,z+1-i,y_1) \vee item(u,z-z_2+1-i,y_1)),$$
$$length(x_1,z_1),\ length(y_1,z_2),$$
$$append(x_1,x_2,x),\ append(y_1,y_2,y)$$

$$\leftarrow length(x,z_1+z_2),\ (\forall ui)(item(u,i,x_1) \vee item(u,i-z_1,x_2) \leftrightarrow item(u,z_1+z_2+1-i,y_1) \vee item(u,z_1+1-i,y_2)),$$
$$length(x_1,z_1),\ length(y_1,z_2),$$
$$append(x_1,x_2,x),\ append(y_1,y_2,y)$$

[by making the obvious instantiation $z := z_1+z_2$
to conserve length in the decompositions of x and y]

$$\leftarrow length(x,z_1+z_2),\ (\forall ui)(item(u,i,x_1) \leftrightarrow item(u,z_1+1-i,y_2)),$$
$$(\forall uj)(item(u,j,x_2) \leftrightarrow item(u,z_2+1-j,y_1)),$$
and calls to length and append as above

[instantiating $i := j + z_1$ *to simplify the arithmetic of the indices, followed by distribution of* $\leftrightarrow$ *through* $\vee$*]*

$$\leftarrow length(x,z_1+z_2),\ reverse(x_1,y_2),\ append(x_1,x_2,x),$$
$$reverse(x_2,y_1),\ append(y_1,y_2,y)$$

[modus tollens by invoking reverse specification twice]

$$\leftarrow reverse(x_1,y_2),\ append(x_1,x_2,x),$$
$$reverse(x_2,y_1),\ append(y_1,y_2,y)$$

[by prefixing the first call with an existential quantifier over the length of x, and then modus tollens invoking A1]

Hence this rather unbeautiful derivation establishes the recursive *reverse* procedure :-

$$reverse(x,y) \leftarrow append(x_1,x_2,x),\ reverse(x_1,y_2),\ reverse(x_2,y_1),\ append(y_1,y_2,y)$$

The proof of this procedure was pursued as the consequence of an assumption that both x and y were decomposable by calls to *append*. The remaining cases to be considered are those arguments in a call to *reverse* which are not so decomposable. Some easy trivializations of the *reverse* specification provide suitable basis procedures, which are stated here without proof :-

$$reverse(x,y) \leftarrow length(x,0),\ length(y,0)$$
$$reverse(x,y) \leftarrow length(x,1),\ length(y,1),\ item(u,1,x),\ item(u,1,y)$$

In order to show that the three derivations providing this procedure set were complete for *reverse* as specified by S, it would be sufficient to prove from S the theorem :-

$$length(x,0) \vee length(x,1) \vee (\exists x_1 x_2)\, append(x_1,x_2,x)$$

thus ensuring that the set of conditions summoned for making conditional substitutions into the goals was exhausted by this particular synthesis. Proof of that theorem is tedious but conceptually simple, and is omitted here.

The behaviour of this procedure set is sufficiently well-known to restrict our discussion of it here to the simple observation that it is inherently recursive for all input-output permutations of its invoking arguments. There are no special virtues manifested by it, except in its role as a general theorem about *reverse* and *append* which can be specialized to give more computationally useful reversal programs. This possibility is the subject of the next discussion.

Iterative Reversal Programs

There are a number of ways of obtaining iterative reversal programs by specializing the recursive procedure set above. Beginning with the simplest way, a simple inspection of the definitions of the *append* and *append** relations is sufficient to establish the sentence below, which becomes an *S-conditional-equivalence* when the *append** definition is added to the specification set. Note that *append** is just a specialization of *append*, so that its substitution for *append* in the above procedures will naturally specialize them as well :-

$$(\ append(x_1,x_2,x) \leftrightarrow append^*(u,x_2,x)) \leftarrow length(x_1,1),\ item(u,1,x_1)$$

As has been demonstrated in the derivation of the lemma, the inference rules which have been named as *S-equivalence* and *S-conditional-equivalence substitutions* are applicable to any sentences as well as to derived goals. Therefore apply the last sentence to the recursive *reverse* procedure to obtain :-

$$reverse(x,y) \leftarrow append^*(u,x_2,x),\ reverse(x_1,y_2),\ reverse(x_2,y_1),\ append(y_1,y_2,y),\ length(x_1,1),\ item(u,1,x_1)$$

When x_1 is specialized to a unit list in this way, the second call can obviously be solved using the second *reverse* basis; so, invoking that basis in response to the second call will transform the above procedure to :-

$$reverse(x,y) \leftarrow append^*(u,x_2,x),\ reverse(x_2,y_1),\ append(y_1,y_2,y),\ length(x_1,1),\ length(y_2,1),\ item(u,1,x_1),\ item(u,1,y_2)$$

[with a slight re-ordering of calls]

$$\vdash_S reverse(u.x_2,y) \leftarrow append(y_1,u.nil,y),\ reverse(x_2,y_1)$$

[by invoking the theorems :-

$$length(u.nil,1) \leftarrow$$
$$item(u,1,u.x_2) \leftarrow$$
$$append^*(u,x_2,u.x_2) \leftarrow$$

trivially implied by S after specifying item over terms, and thus macroprocessing out the high-level selectors]

Now invocation of this procedure can be implemented quasi-iteratively when *y* is given as input, since the call to *append* can then be solved deterministically before activating the call to *reverse*. Thus to iteratively reverse some list such as *a.b.c.nil* the invoking goal can be chosen as $\leftarrow reverse(x,a.b.c.nil)$. A sufficient basis to accompany this procedure is the macroprocessed basis given previously for dealing with empty lists :-

$$reverse(nil,nil) \leftarrow$$

The procedure set consisting of these two procedures for lists represented by terms, and specialized so as to split the list *x* into its first member and its 'rest', is adequate if some means can be found of solving the call to *append* efficiently. The orthodox *append* procedures are of little use here because they cannot directly access the last member *u* of the argument *y* in the call $append(y_1,u.nil,y)$.

The problem of computing the last member of a list as just discussed vanishes if an alternative data structure representation is chosen. For instance, in Section 3.2 the term $t(u,x,v)$ was used to represent a list with first member u, last member v and middle x. There it was shown that this representation allowed an iterative computation which, at the level of the source program at least, appeared to have direct access to all the components of the list submitted as input for reversal. How efficient that arrangement would really be in practice would depend upon how the interpreter stored and matched terms constructed using the t-function.

An even easier way of dealing with the call to *append* in the above procedure is to represent lists by terms of the form $app(y_1,y_2)$ such that $app(y_1,y_2)$ represents the list obtained by appending y_2 to y_1. By defining the meaning of *item* for this representation as follows :-

$$item(u,i,app(nil,nil)) \leftrightarrow false$$

$$item(u,i,app(y_1,y_2)) \leftrightarrow item(u,i,y_1) \vee (\exists w)(length(y_1,w), item(u,i-w,y_2))$$

and assigning these axioms to S, it is then easy to derive from S the theorem :-

$$append(y_1,y_2,app(y_1,y_2)) \leftarrow$$

and so macroprocess out the call to *append* , leaving an iteratively invokable *reverse* procedure :-

$$reverse(u.x_2,app(y_1,u.nil)) \leftarrow reverse(x_2,y_1)$$

The efficiency of an implementation of this procedure would depend critically upon the interpreter's capacity to perform clever evaluations in the binding environment in order to manage the concrete data expressed at source-level by these rather clumsy terms. However, without the use of interpreters capable of efficient management of such terms, none of the procedure sets considered here for *reverse* is satisfactory. This is why we now turn attention to the derivation of procedures for the more sophisticated *reverse** relation, which was also examined in Chapter 3.

The *reverse** predicate has been specified by Robert Kowalski *[Lecture Notes, Syracuse, 1978]* using the sentence :-

$$reverse^*(z,x,y) \leftrightarrow (\exists w)(reverse(x,w),\ append(w,z,y))$$

so that $reverse^*(z,x,y)$ holds when y is the result of appending z to the reverse of x. Thus in order to compute y as the reverse of x it is sufficient to solve $reverse^*(z,x,y)$ for the case where z is the empty list. A formal proof of this would only require a lemma :-

$$(append(w,z,y) \leftrightarrow y=w) \leftarrow length(z,0)$$

to be derived using the *append* specification. Then an immediate substitution for the *append* predicate in the *reverse** specification would establish the first procedure in our desired procedure set, this being :-

$$reverse(x,y) \leftarrow reverse^*(nil,x,y) \quad [using \ \ length(nil,0)\leftarrow]$$

The derivation of this procedure then constitutes a synthesis for the *reverse* relation having *reverse** as its primitive. Therefore the next level in the synthesis hierarchy has the aim of synthesizing a procedure set for *reverse**. Only a brief outline of this is shown below.

Two lemmas are required in order to derive the recursive *reverse** procedure :-

$$append(w',u.z,y) \leftrightarrow (\exists w)(append(w',u.nil,w),\ append(w,z,y))$$
$$reverse(u.x_2,w) \leftarrow reverse(x_2,w'),\ append(w',u.nil,w)$$

The first of these is derived from the properties of the *append* relation and just expresses a consequence of its associativity. The second lemma is just the recursive *reverse* procedure already derived; note that the derivation of more sophisticated procedures often relies upon the use of simpler procedures treated as specification axioms in *S*. From these preliminaries, the following treatment is straightforward :-

$\leftarrow reverse^*(z,x,y)$

$\leftarrow reverse(x,w),\ append(w,z,y)$

$\leftarrow reverse(x_2,w'),\ append(w',u.nil,w),\ append(w,z,y)$

[first call invokes the reverse procedure above, inducing the binding $x:=u.x_2$*]*

$\leftarrow reverse(x_2,w'),\ (\exists w)(append(w',u.nil,w),\ append(w,z,y))$

[prefixing last two calls by $(\exists w)$*]*

← *reverse*(x_2,*w'*), *append*(*w'*,*u.z*,*y*) *[modus tollens, invoking append lemma]*

← (∃*w'*)(*reverse*(x_2,*w'*), *append*(*w'*,*u.z*,*y*)) *[prefixing by (∃w')]*

← *reverse**(*u.z*,x_2,*y*) *[modus tollens using reverse* spec.]*

and so we infer the procedure :-

*reverse**(*z*,*u.*x_2,*y*) ← *reverse**(*u.z*,x_2,*y*)

Inspection of the second inference step in this derivation (the one which invoked the associative property of *append*) shows that it introduces the assumption that *x* is decomposable, that is, has the structure *u.*x_2. An alternative branch at that point deals with the other case where *x* is the empty list. From the procedure set for the recursive reverse algorithm we have the two bases :-

reverse(*nil*,*nil*) ←

append(*nil*,*y*,*y*) ←

which can now be invoked as lemmas in the derivation below which provides a basis for *reverse** :-

← *reverse**(*z*,*x*,*y*)

← *reverse*(*x*,*w*), *append*(*w*,*z*,*y*) *[just proceeding as before]*

□ *[taking the other branch by invoking the bases for append and reverse, thereby inducing the bindings x:=nil, w:=nil and z:=y]*

thus inferring the basis :-

*reverse**(*y*,*nil*,*y*) ←

The properties of this new procedure set have already been discussed in Chapter 3 and so do not warrant further discussion here, other than to recall the iterative behaviour which they give; this will be reasonably efficient if the interpreter possesses good means of storing and accessing the terms constructible from . and *nil*.

The *reverse** derivation closely parallels that of the 3-place factorial program described in Clark's paper (12); there, he exploits the associativity of multiplication where above we use the same property of the appending operation. Clark also derives procedures for Kowalski's 4-place factorial relation, whose top-down execution simulates the bottom-up execution of the conventional 2-place program.

Similarly it is possible to derive a procedure set which interrogates a 4-place relation *reverse*** :-

$$reverse(x,y) \leftarrow reverse^{**}(x,y,nil,nil)$$
$$reverse^{**}(x,y,x,y) \leftarrow$$
$$reverse^{**}(x,y,w,z) \leftarrow append(x',u.w,x),\ append(z,u.nil,z'),\ reverse^{**}(x,y,u.w,z')$$

whose top-down execution behaves like the bottom-up execution of the procedures which use the 2-place *reverse* relation :-

$$reverse(nil,nil) \leftarrow$$
$$reverse(u.x',y) \leftarrow reverse(x',y'),\ append(y',u.nil,y)$$

Given the goal $\leftarrow reverse(a.b.c.nil,y)$ both computations compute the successive approximations *nil*, *c.nil*, *c.b.nil* and *c.b.a.nil* to the output variable *y*. By contrast the 3-place program just examined computes the successive approximations *nil*, *a.nil*, *b.a.nil* and *c.b.a.nil* when executed top-down.

By analogy to the procedure set investigating ordered lists which was derived in Section 5.5, there exists a reversal program which maintains explicit pointers in the procedures' arguments in order to allow direct access to the individual members of the lists in question. Its derivation is sufficiently similar to that of the analogous orderedness program to allow its omission here, but it is interesting to see just what that program looks like. Suppose that two lists L_1 and L_2 are given as input and represented in such a way as to allow direct look-up of any i^{th} member (for example, by representing them by sets of *item* assertions). Then the following procedures provide excellent top-down behaviour when executed for the goal $\leftarrow reverse(L_1,L_2)$:-

$$reverse(x,y) \leftarrow length(x,0),\ length(y,0)$$
$$reverse(x,y) \leftarrow length(x,j),\ reverse^{\dagger}(x,1,j,y)$$
$$reverse^{\dagger}(x,j,j,y) \leftarrow length(x,j),\ item(u,j,x),\ item(u,1,y)$$
$$reverse^{\dagger}(x,i_1,i_2,y) \leftarrow i_1<i_2,\ item(u,i_1,x),\ length(x,j),\ item(u,j+1-i_1,y),\ reverse^{\dagger}(x,i_1+1,i_2,y)$$

Here the *reverse*† relation is specified by :-

$$reverse^{\dagger}(x,i_1,i_2,y) \leftrightarrow (\exists j)(length(x,j),\ reverse(f(x,i_1,i_2),f(y,j+1-i_2,j+1-i_1)))$$

where the term $f(x,i_1,i_2)$ represents the list fragment of x which extends from its i_1^{th} up to and including the i_2^{th} member. When one list, say y, is required as output given the other as input, the successive activations of the call *item(u,j+1-i,y)* will contribute satisfactorily to a cumulative term representation of the output. Alternatively, if the output y is required to be represented by a set of computed *item* assertions, there may exist semantically justifiable ways of inducing such assertions in order to 'quasi-solve' those calls to *item(u,j+1-i,y)* ; but this is a matter beyond the scope of the present discussion dealing with the logic underlying the reversal algorithms.

In summary it should be noted that the algorithms considered here all pursue the task of accessing j members from one list of length j and then accessing or constructing exactly j members of the other list. The differences between their behaviours are really just associated with different *implementations* of that task, for example, whether they perform it recursively or iteratively, or whether or not they induce a significant binding environment. By contrast, the next section examines a problem in which the opportunity exists for achieving differences in efficiency between alternative programs which are not attributable to superficial differences in implementation.

6.2 : SEARCHING LISTS FOR DUPLICATES

Specifying the Problem

Whereas the problem in the previous section concerned a relation between two lists, here we consider just a *1*-ary relation defined upon a single list. The property expressed by this relation holds when the list in question contains at least one duplicate member; when this is satisfied by some list *x* the predicate *duplic(x)* holds. The *duplic* relation is specified by the following sentence which requires that some member *u* shall occur in both positions *i* and *j* such that *i<j* :-

$$duplic(x) \leftrightarrow (\exists uij)(item(u,i,x),\ item(u,j,x),\ i<j)$$

The initial specification set contains this sentence, the list axioms *A1-A3* and the specification for the *append** relation. General properties of = and < (over non-negative integers) are assumed implicitly. These axioms provide enough information for the derivation of the first two algorithms considered here. It is assumed, as always, that the programs are intended primarily for a Prolog-like interpreter.

The Naive Algorithm

The naive algorithm applies Prolog-like control to a procedure set having just one procedure for *duplic*. This procedure is trivially implied by the *duplic* specification, which, having a definiens consisting of just a conjunction of atoms, immediately provides an executable Horn clause :-

$$duplic(x) \leftarrow item(u,i,x),\ item(u,j,x),\ i<j$$

Now suppose that procedures for solving calls to *item* efficiently are provided; for instance *x* may be represented assertionally, or procedures for *item* may be available for accessing some other representation. Further assume that calls to < can be processed directly by the interpreter. Then the procedures already considered will make up a complete program body for solving a call to duplic.

The computation generated from this algorithm is essentially the iteration of one loop within another, in much the same way as the *pick* program examined previously which executed the procedure :-

pick(u,v,z) ← *u*ε*z*, *v*ε*z*, *u*<*v*

Observe that this *duplic* procedure does not constrain the order in which members are selected from *x* and compared, this being determined instead by the interpreter's strategy for processing the calls to *item*. Assume however that the most straightforward arrangement prevails in which members are selected in order of increasing index. Then the first solution (if any) of *duplic(x)* is obtained by finding the least indices m_1 and m_2 satisfying *(item(u,*m_1*,x), item(u,*m_2*,x),* $m_1 < m_2$*)*.

Let the algorithm described above be designated A1. A convenient measure of its efficiency is the number of comparisons executed between members of the given list *x*, since the essential behaviour of the algorithm is to select some *u* satisfying *item(u,i,x)* and then successively compare it with other members of *x* until a matching member is found with an index *j* exceeding *i*. Assuming that *x* has *n* members, the total comparison count of A1 needed to discover the first solution *i:=*m_1, *j:=*m_2 is *N*(A1) such that :-

$$N(\text{A1}) = m_2 + n(m_1 - 1)$$

This comparison count is rather unsatisfactory in that every pair of members having distinct indices less than m_1 is compared twice; moreover, every member having an index less than or equal to m_1 is compared with itself. Thus there are $\frac{1}{2}(m_1)(m_1-1) + m_1$ redundant comparisons executed by A1 assuming the accessing protocol suggested above for selecting the members of *x*. Note also that *N*(A1) is dependent upon *n*, despite the intuitively obvious fact that the problem can be solved - if at all - by inspecting just the first m_2 members. The remaining algorithms to be considered respectively remedy these two failings in the efficiency of algorithm A1.

The Improved Naive Algorithm

The reason for the redundant comparisons executed by A1 is that the logic does not constrain the order in which members are selected and compared. In particular the algorithm does not take into account the fact that if the *i*th member *u* fails to match some *j*th member *v*, then there is no point in attempting at some later time to select that same *v* as the *i*th member and then compare it with a *j*th member *u*. Instead it generates comparisons for the index pair *(i,j)* and the index pair *(j,i)* without recognizing that these compare the same

members. The improved naive algorithm, designated A2, constrains the scheduling of the choices of comparison indices *(i,j)* such that these redundancies cannot arise. The derivation of *duplic* procedures for A2 requires just the minimal knowledge about the constructibility of lists as expressed by axioms *A1-A3* together with the *append** specification. By introducing calls to *append**, we can arrange that the logic deals explicitly with the components of *x* so as to logically preclude redundant comparisons. The obvious agency for introducing such calls is the *S-conditional-equivalence substitution*, as is now demonstrated.

Assuming the specification set already enunciated, consider a derivation for *duplic* :-

← *duplic(x)*

← *item(u,i,x), item(u,j,x), i<j*

← *(item(u,i-1,x') ∨ (u=u',i=1)),*
(item(u,j-1,x') ∨ (u=u',j=1)), i<j, append(u',x',x)*

[orthodox S-cond.-equiv. substitution, replacing two item predicates conditional upon a call to append]*

A little intuition is useful at this point in order to decide how to simplify this goal. Note that if the goal is to be solvable then the solutions for *i* and *j* must satisfy either *(i>1, j>1)* or *(i=1, j>1)* as a consequence of the general properties of <. Considering the former case first, its application to the current goal simplifies the first two calls, as follows :-

← *(item(u,i-1,x') ∨ false),*
(item(u,j-1,x') ∨ false), i<j, append(u',x',x), i>1, j>1*

[because the properties of = and > S-imply the S-conditional-equivalence :-

(i=1 ↔ false) ← i>1]

← *item(u,i-1,x'), item(u,j-1,x'), i<j, append*(u',x',x)*

[simplifying first two calls, and deleting last two calls as they are then implied by the first two]

← *item(u,$\hat{i}$,x'), item(u,$\hat{j}$,x'), $\hat{i}$+1<$\hat{j}$+1, append*(u',x',x)*

[instantiating $\hat{i}$:=i-1, $\hat{j}$:=j-1]

← *$(\exists u\hat{i}\hat{j})$(item(u,$\hat{i}$,x'), item(u,$\hat{j}$,x'), $\hat{i}<\hat{j}$, append*(u',x',x)*

[exploiting properties of <, and prefixing with $(\exists u\hat{i}\hat{j})$]

← *duplic(x'), append*(u',x',x)* *[modus tollens using duplic spec.]*

Returning to the earlier branch point, the consequence of choosing the alternative case (*i=1, j>1*) is the derivation :-

← *(item(u,i-1,x') ∨ (u=u',i=1)),*
(item(u,j-1,x') ∨ (u=u',j=1)), i<j, append(u',x',x)*

[the goal at the branch point again]

← *(false ∨ u=u'),*
(item(u,j-1,x') ∨ (u=u',false)) ,i<j, append(u',x',x), i=1, j>1*

[substitutions conditional upon i=1, j>1,
and using list axioms to assert indices must be ⩾1]

← *item(u',$\hat{j}$,x'), append*(u',x',x)*

[some obvious simplifications, and instantiation $\hat{j}$:=j-1]

These two derivations comprise a complete synthesis for the *duplic* relation because they exhaust the two cases for the values of *i* and *j*. Note that they both consider the case where *x* is decomposable; there is no case corresponding to an empty list *x*, since the assumption that *x* was empty would make the derivation goal unsolvable. The two procedures inferred in the synthesis are :-

duplic(x) ← append(u',x',x), item(u',j,x')*
duplic(x) ← append(u',x',x), duplic(x')*

This is the procedure set for algorithm A2. Now suppose that in a Prolog-like execution the first procedure is invoked and fails. Then *u'* will have been compared with all members of *x'*. When control passes to the second procedure, the *append** call effectively discards this instance of *u'* from the ensuing computation, so that no subsequent comparisons of members with that instance can be executed. Thus the calls to *append** eliminate the possibility of redundant comparisons. The comparison count of algorithm A2 when it terminates successfully is :-

$$N(\text{A2}) = (m_2 - m_1) + (m_1 - 1)(n - \tfrac{1}{2}m_1)$$

The difference between $N(A1)$ and $N(A2)$ is just the number $m_1 + \frac{1}{2}(m_1)(m_1-1)$ of redundant comparisons executed by A1.

Both A1 and A2 generate comparisons of members whose indices fall in the range $m_2 \leq i \leq n$. As a result, the measures $N(A1)$ and $N(A2)$ both depend upon n. This can result in some very inefficient computations for certain choices of goal. For example, if n is very large and both m_1 and m_2 are small relative to n, then both $N(A1)$ and $N(A2)$ approximate to $(m_1-1)n$, so that both have comparison counts of order n even when the matching pair could be found by just inspecting the first few (m_2) members of x. The next algorithm A3 performs exactly that inspection in order to solve *duplic(x)*.

The Length-Independent Algorithm

The logic underlying the length-independent algorithm A3 can be anticipated by the following intuitive reasoning. Suppose at some instant in the execution of A3 the fragment $(x_1, \ldots, x_k)$ of x has been inspected and found to have no duplicates. Moreover, assume that *duplic(x)* is solvable with $i:=m_1$, $j:=m_2$. Then k is certainly less than m_2. Now consider the two cases $k<m_1$ and $k \geq m_1$. If $k<m_1$ then the duplicate members must occur in the fragment $(x_{k+1}, \ldots, x_n)$; if $k \geq m_1$ then one of the duplicate members occurs in $(x_1, \ldots, x_k)$ whilst the other occurs in $(x_{k+1}, \ldots, x_n)$. A convenient formalization of these ideas makes use of a new predicate whose arguments represent the fragments explicitly; let $duplic^*(z_1,z_2)$ hold when z_1 is some permutation of $\{x_1, \ldots, x_k\}$ and z_2 is the fragment $(x_{k+1}, \ldots, x_n)$. Informally, z_1 is associated with the set of members currently known to be distinct, whilst z_2 is the remaining fragment of the input list which still awaits inspection for duplicates. The logical specification of *duplic** is :-

$$duplic^*(z_1,z_2) \leftrightarrow (\exists uij)(item(u,i,z_1),\ item(u,j,z_2)) \vee duplic(z_2)$$

Another way of informally considering the meaning of $duplic^*(z_1,z_2)$ is to say that the structure (z_1,z_2) contains duplicates but z_1 does not; in which case either some member is common to z_1 and z_2, or else some member has duplicate occurrences in z_2.

The objective now is, firstly to find a way of solving a call to *duplic* using procedures for *duplic** and, secondly, to derive procedures for *duplic** which can be controlled so as to give A3.

In order to make use of the *duplic** predicate in the solution of *duplic(x)* it is useful to recall the list axiom *A3* :-

$$length(z,w) \leftrightarrow (\forall i)(1 \leqslant i \leqslant w \leftrightarrow (\exists u) item(u,i,z))$$

Both *A3* and the *duplic** specification are assumed to be established in *S*, and can now be combined as follows : using *A3* to make an *S-cond.-equiv. substitution*, substitute for the predicate $item(u,i,z_1)$ conditional upon the assumption $length(z_1,0)$. This gives the sentence :-

$$(duplic^*(z_1,z_2) \leftrightarrow (\exists uij)(false,\ item(u,j,z_2)) \vee duplic(z_2)) \leftarrow length(z_1,0)$$

which, after some trivial simplification, clearly implies the following procedure for *duplic* :-

$$duplic(z_2) \leftarrow duplic^*(z_1,z_2),\ length(z_1,0)$$

Renaming the variables and ordering the calls appropriately, this gives the first procedure in the program body for algorithm A3 :-

$$duplic(x) \leftarrow length(z_1,0),\ duplic^*(z_1,x)$$

The next step is a synthesis for *duplic**. The derivation tree for this obviously begins as follows :-

$\leftarrow duplic^*(z_1,z_2)$
$\leftarrow (\exists uij)(item(u,i,z_1),\ item(u,j,z_2)) \vee duplic(z_2)$

[note that this has just one call - a disjunction]

We shall now make some rather subtle substitutions into this goal using a number of lemmas. The overall objective here is to capture the algorithmic notion of successively selecting members from the fragment z_2 and testing them for membership in z_1. This motivates the search for lemmas which allow useful goal substitutions conditional upon $append^*(u',z_2',z_2)$.

Note firstly that the two derived *duplic* procedures can be expressed as :-

$$(duplic(x) \leftarrow item(u',j,x')) \leftarrow append^*(u',x',x)$$
$$(duplic(x) \leftarrow duplic(x')) \leftarrow append^*(u',x',x)$$

(that is, the *duplic* procedures used in the improved naive algorithm.)

Then the completeness of that procedure set for *duplic* determines that these two sentences can be combined to provide a single lemma :-

$$(duplic(z_2) \leftrightarrow (\exists j)item(u',j,z_2') \vee duplic(z_2')) \leftarrow append^*(u',z_2',z_2)$$

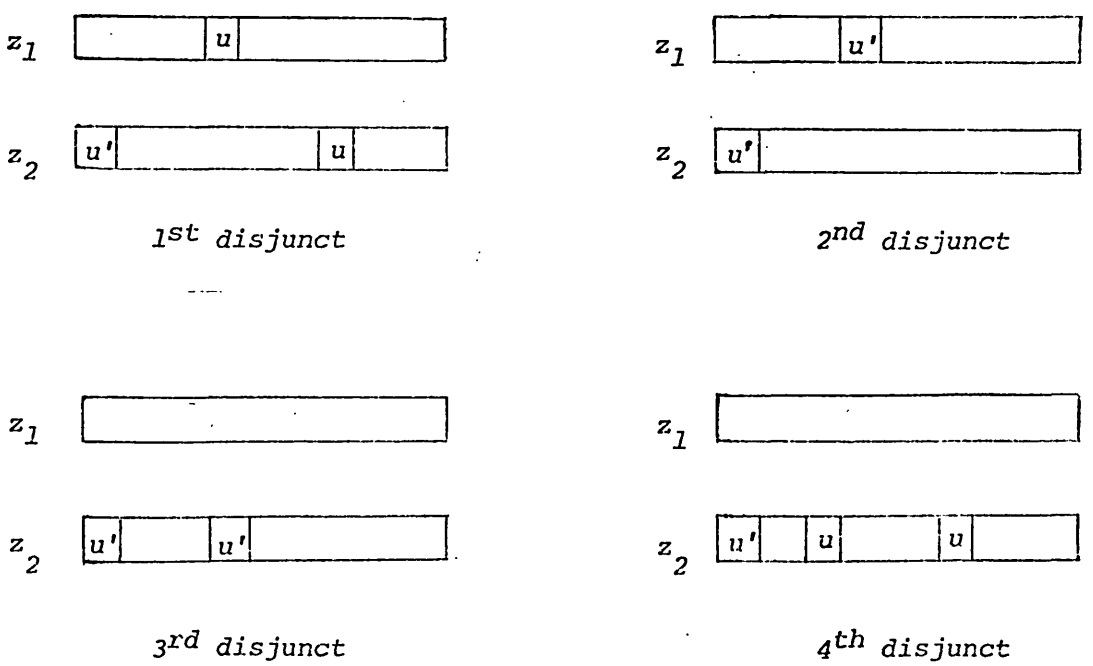
From the specification of *append** we also have the lemma :-

$$(\forall uj)(item(u,j,z_2) \leftrightarrow item(u,j-1,z_2') \vee (u=u',j=1)) \leftarrow append^*(u',z_2',z_2)$$

The two lemmas can now be used to make *S-conditional-equivalence substitutions* for the respective goal subformulas $duplic(z_2)$ and $item(u,j,z_2)$, the result of which after a little simplification by distribution is :-

$$\begin{aligned} \leftarrow\ & ((\exists ui)(item(u,i,z_1), item(u,j-1,z_2')) \\ & \vee (\exists i)item(u',i,z_1) \\ & \vee (\exists j)item(u',j,z_2') \\ & \vee duplic(z_2')), append^*(u',z_2',z_2) \end{aligned}$$

The disjuncts of the first call just represent the four cases in which $(z_1,u'.z_2')$ may contain duplicates when z_1 does not. It may be helpful to see these cases portrayed below :-



Suppose now that the member u' is selected from z_2 and is found not to belong to z_1. Then the only way in which the duplicate problem

can still be solved is by showing that either z_2' contains duplicates or that z_2' contains a member identical to one of those already inspected by the algorithm - these being u' together with the members now in z_1. An obvious step for the algorithm to take is therefore to transfer u' from z_2 to z_1, producing z_2' and z_1'' respectively. Consider each of the disjuncts of the goal's first call on the assumption that u' is to be transferred in this way. Subsequent solution of the problem depends upon finding some state of the two list fragments amongst (z_1'', z_2') and its successors arising from further transfers such that in that state there is some member common to both fragments. The 2nd and 4th disjuncts express an arrangement in which the question of a common member occurring in succeeding fragments after transferring u' remains undecided. By contrast, the 1st and 3rd disjuncts express arrangements in which a member common to both fragments is assured after transferring u'. With this latter remark in mind, it is quite easy to show that the disjunction of the 1st and 3rd disjuncts can be substituted by the formula $(\exists uij)(item(u,i,z_1''), item(u,j,z_2'))$ which expresses the fact that the new fragments must have a common member. This substitution is an *S-conditional-equivalence substitution* whose condition is $append^*(u',z_2',z_2), append^*(u',z_1,z_1'')$. This transforms the goal to :-

$$\leftarrow ((\exists uij)(item(u,i,z_1''), item(u,j,z_2')) \vee (\exists i)item(u',i,z_1) \vee duplic(z_2')), append^*(u',z_2',z_2), append^*(u',z_1,z_1'')$$

$$\leftarrow (duplic^*(z_1'',z_2') \vee (\exists i)item(u',i,z_1)), append^*(u',z_2',z_2), append^*(u',z_1,z_1'')$$

[modus tollens by invoking duplic spec. in order to summarize the ways of solving the problem after transferring u']*

This goal now permits two procedures for *duplic** to be inferred :-

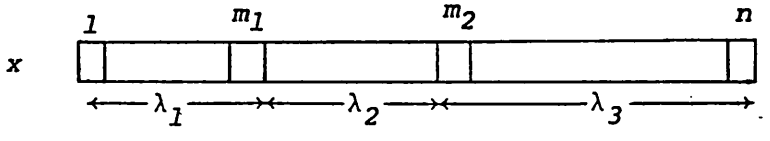
$$duplic^*(z_1,z_2) \leftarrow append^*(u',z_2',z_2), item(u',i,z_1)$$
$$duplic^*(z_1,z_2) \leftarrow append^*(u',z_2',z_2), append^*(u',z_1,z_1''), duplic^*(z_1'',z_2')$$

These form a complete procedure set for *duplic** because the goal transformations which were made all preserved *S-equivalence*

conditional upon the calls to *append**. The last condition is exhaustive in that the alternative assumption that z_2 was not decomposable would make the goal unsolvable. Assuming then that procedures are already available for solving calls to *item* and *append**, the synthesis of a complete program body is complete. Executing a call to *duplic* using the three derived procedures and Prolog-like control gives algorithm A3. As execution proceeds, successive distinct members are deleted from the list z_2 and stacked in the record z_1; z_1 is initially empty. Each new member is deposited in z_1 by the second *duplic** procedure after the first *duplic** procedure has failed to show that this member already occurs in z_1. If, however, a call to the first *duplic** procedure is successful then computation terminates; at this point, exactly (m_2-1) members which were originally in z_2 have been transferred to z_1, and one of them is now known to match the m_2^{th} member of z_2. The total number of comparisons made by A3 in order to find the first solution of *duplic(x)* is :-

$$N(\text{A3}) = (m_2 - m_1) + (m_2 - 1)(m_2 - 2)$$

which is independent of n. Algorithm A3 is not necessarily more efficient than A2 in terms of comparison counts. If λ_1, λ_2 and λ_3 are the lengths depicted below :-



then we have $N(\text{A3}) \leqslant N(\text{A2})$ if and only if $\lambda_2(\lambda_2+1) \leqslant 2\lambda_3(\lambda_1-1)$.

The essential differences between the three algorithms can be summed up by saying that every inspected member of x is compared in A1 with its predecessors, itself and its successors; in A2 with its successors only; and in A3 with its predecessors only. Algorithm A1 is therefore less efficient than both A2 and A3.

Since algorithm A3 has no other task than to select successive members from x and compare them with their predecessors, it would seem unnecessary to implement that process using two distinct data structures z_1 and z_2. Those data structures are useful for describing the problem abstractly as in the *duplic** derivations above, but could

clearly be implemented implicitly instead by just employing a suitable pointer system governing selections and comparisons in x. In fact all we need is one pointer j varying from 1 up to n which marks the boundary between the fragments z_1 and z_2. Then a transformation similar to those shown earlier for the palindrome and orderedness problems gives the rather elegant alternative logic component for algorithm A3 :-

$$duplic(x) \leftarrow duplic^{\dagger}(x,1)$$

$$duplic^{\dagger}(x,j) \leftarrow item(u,j,x),\ find(u,1,j-1,x)$$
$$duplic^{\dagger}(x,j) \leftarrow length(x,w),\ j<w,\ duplic^{\dagger}(x,j+1)$$

$$find(u,i,j,x) \leftarrow i \leqslant j,\ item(u,i,x)$$
$$find(u,i,j,x) \leftarrow i<j,\ find(u,i+1,j,x)$$

This gives excellent behaviour for an implementation allowing direct access to the members of x selected by the pointer j in the calls to *item*. The relations introduced above may be specified informally using the list membership predicate $\hat{\varepsilon}$ and the f-notation used in previous examples for denoting list fragments :-

$$duplic^{\dagger}(x,j) \leftrightarrow x_j \hat{\varepsilon} f(x,1,j-1) \vee (\exists w)(length(x,w),\ duplic(f(x,j,w)))$$

$$find(u,i,j,x) \leftrightarrow u \hat{\varepsilon} f(x,i,j)$$

6.3 : GENERATION OF FACTORIAL TABLES

Specifying the Problem

The problem considered in this section is that of constructing a table of factorials containing entries $(u,u!)$ for $u = 0, \ldots, z$ where z is some non-negative integer given as input. Such a table might serve as a useful data structure accessed by some other program requiring frequent look-up of a limited range of factorials. Several programs for computing individual factorials are discussed in Clark's paper (12) and are quite interesting as demonstrations of various logic programming styles; however, the task of constructing an entire series of factorials is more interesting in that it provides scope for varying the exploitation of their dependencies and order of generation, as well as raising matters of data structure representation. Construction of factorial tables is also briefly considered in the paper by Burstall and Darlington (10) who show how to compute them using programs represented as sets of recursive function definitions.

The logical specification of the problem is accomplished quite concisely with the sentences :-

$$table(x,z) \leftrightarrow (\forall uv)(entry(u,v,x) \leftrightarrow 0 \leqslant u \leqslant z,\ fact(u,v))$$
$$fact(u,v) \leftrightarrow (u=0,\ v=1) \vee (\exists w)(times(u,w,v),\ fact(u-1,w))$$

in which $table(x,z)$ holds when x is a table containing entries $(0,0!), \ldots, (z,z!)$; $entry(u,v,x)$ holds when (u,v) is an individual entry in table x ; $times(u,w,v)$ expresses the multiplication relation over non-negative integers ; and $fact(u,v)$ expresses $u!=v$. Elementary properties of $\leqslant$ over the latter domain are implicitly assumed in S as always, and the only property of $times(u,w,v)$ which we shall need to summon is that any pair (u,v) determines w uniquely when that predicate is satisfied. Any calls to $\leqslant$ and $times$ which may appear in the derived procedures will be assumed to be directly executable by the intended Prolog-like interpreter.

Several algorithms are presented here which are classified according to their arithmetical properties (measured by how many multiplications are needed to construct a table of a given size) and according to the order in which they generate the table entries.

When an algorithm generates entries in the order $(0,0!)$, ..., $(z,z!)$ we shall call it a *natural ordering* algorithm, and when it generates them in the reverse order we shall call it an *anti-natural ordering* algorithm. Five algorithms are considered altogether, and presented in order of decreasing naivety.

Quadratic Anti-natural Ordering Algorithm

This simple algorithm is derived by exploiting some basic ideas about the constructibility of the data structures which we have called 'tables'. Any given table can be viewed as simply a set of pairs ('entries') and so be expressed in terms of the set union of its component subsets. Here it is convenient to regard the table as the union of a singleton containing some entry (u',v') with the set X' of all its other entries. Then the construction of tables can be specified by the sentence below which is admitted to the specification set :-

$$enter(u',v',x',x) \leftrightarrow (\forall uv)(entry(u,v,x) \leftrightarrow (u=u',v=v') \vee entry(u,v,x'))$$

where $enter(u',v',x',x)$ expresses $x = \{(u',v')\} \cup x'$ and is named so as to reflect the notion of an algorithm which successively 'enters' the computed entries into some partially constructed table.

The derivation below uses this knowledge about the table's structure to make a substitution for the specification's predicate $entry(u,v,x)$ in order to explicate the way in which any particular computed entry is assigned to the table. The result of this is that the table x becomes expressed in terms of its sub-table x'; to achieve this objective it is necessary to exploit the fact that the size of x' is one less than that of x, which is easily ensured, as will be seen in the derivation, by appealing to an elementary property of $\leqslant$ assumed in S :-

$$0 \leqslant u \leqslant z \leftrightarrow u=z \vee 0 \leqslant u \leqslant z-1$$

With these preliminaries established, the derivation proceeds :-

$\leftarrow table(x,z)$

$\leftarrow (\forall uv)(entry(u,v,x) \leftrightarrow 0 \leqslant u \leqslant z,\ fact(u,v))$

$\leftarrow (\forall uv)(entry(u,v,x') \vee (u=u',v=v') \leftrightarrow 0 \leqslant u \leqslant z,\ fact(u,v)),\ enter(u',v',x',x)$

[S-cond.-equiv. substitution for the entry predicate]

$\leftarrow (\forall uv)(entry(u,v,x') \leftrightarrow 0 \leqslant u \leqslant z-1,\ fact(u,v)),$
$(\forall v)(\ v=v' \leftrightarrow fact(z,v)),\ enter(z,v',x',x)$

[S-equiv. substitution for the predicate $0 \leqslant u \leqslant z$ using the axiom about $\leqslant$ declared above; then assume $u' := z$ to reflect the anti-natural ordering determined by arranging that the computed entry (u,v) is $(z,z!)$; then distribute $\leftrightarrow$ through $\vee$ and simplify ; then distribute $\vee$ through conjunction]

$\leftarrow table(x',z-1),(\forall v)(v=v' \leftrightarrow fact(z,v)),\ enter(z,v',x',x)$

[modus tollens]

The interesting problem now arises of the significance of the goal's second call, and how to process it. The call expresses the requirement of showing that the factorial of any u is unique, and arises in the derivation because the *table* specification in S quantifies u and v in such a way as to admit the possibility of arbitrarily many instances of v satisfying $fact(u,v)$ for a given u. The other specification axioms do not explicitly preclude this possibility. Here we shall assume that S contains enough knowledge about = and *times* to prove the uniqueness of any factorial as a lemma; summoning that lemma will then replace the second call to leave :-

$\leftarrow table(x',z-1),\ fact(z,v'),\ enter(z,v',x',x)$

The calls are now clearly all atomic. Suppose now that we backtrack through the derivation to the point at which it was decided to introduce considerations of the constructibility of x. Here there is no point in trying instead the usual assumption that x can be an empty table, because - as a little experimentation will quickly confirm - there is then no simplification which will produce a solvable goal; the *table* specification insists that any table must contain at least one entry $(0,0!)$. Nevertheless we certainly require a basis for the recursive *table* procedure inferred from the above derivation. Clearly a sufficient basis is that which deals with the most trivial table, namely the unit table $\{(0,0!)\}$ specified by :-

$unit\text{-}table(x,u',v') \leftrightarrow (\forall uv)(entry(u,v,x) \leftrightarrow u=u',\ v=v')$

Admitting this to S and then pursuing the alternative derivation dealing with the case of a unit table, the procedure set for *table* concluded from this analysis is :-

table(x,0) ← unit-table(x,0,1)
table(x,z) ← fact(z,v'), table(x',z-1), enter(z,v',x',x)

This set is complete for *table*. Now it remains to provide means of solving calls to *fact*, *enter* and *unit-table*. Now *S* already contains sufficient information for the synthesis of a complete procedure set for *fact*. The *fact* procedures which are most easily derived here are the following, which are both trivially implied by the *fact* specification :-

fact(0,1) ←
fact(u,v) ← fact(u-1,w), times(u,w,v)

These just represent the conventional top-down recursive computation of factorials. As an alternative to these we could instead make use of the more efficient procedure sets derived by Clark (12) ; however, all these procedure sets have the property that their computations give rise to *z* multiplications in the course of computing any *z!* for *z>0*.

If some procedure set is devised for solving calls to *enter*, leaving the call *enter(z,v',x',x)* in the *table* procedure above as it stands, then that latter procedure must clearly be invoked recursively. On the other hand, a choice of table representation which permitted that call to be eliminated by macroprocessing would then allow iterative invocation. In view of this, admit now to *S* the low-level data-accessing axioms for the simplest term representation :-

entry(u,v,∅) ↔ false
entry(u,v,e(u',v'):x') ↔ (u=u',v=v') ∨ entry(u,v,x')

which employs ∅ and : to construct sets of pairs constructed from *e*. Then *S* will trivially imply the data-accessing procedures :-

unit-table(e(u',v'):∅, u', v') ←
enter(u',v',x', e(u',v'):x') ←

and thus enable the procedures for *table* to be macroprocessed, giving :-

table(e(0,1):∅, 0) ←
table(e(z,v'):x', z) ← fact(z,v'), table(x',z-1)
fact(0,1) ←
fact(u,v) ← fact(u-1,w), times(u,w,v)

These procedures instigate $\frac{1}{2}(z)(z+1)$ multiplications in the course of solving a goal $\leftarrow table(x,z)$ given an input instance $z>0$. This holds irrespective of the fact that a particular representation has been chosen for the computed table. Assuming commitment to Prolog's control strategy, the quadratic dependence of the algorithm's arithmetic burden upon z arises solely by virtue of the fact that z independent factorials have to be computed by the explicit calls to *fact* in the *table* procedure. We should obviously be able to improve upon this; for instance, $z!$ and $(z-1)!$ are computed here by a total of $(2z-1)$ multiplications even though $z!$ is computable in principle by just one multiplication once $(z-1)!$ is known. This latter consideration is the intuition underlying the next algorithm.

Linear Anti-natural Ordering Algorithm

A more efficient algorithm can be obtained using the axioms in S together with the procedures already derived. Recalling the recursive procedures for *table* and *fact* :-

$$table(x,z) \leftarrow fact(z,v'),\ table(x',z-1),\ enter(z,v',x',x)$$
$$fact(u,v) \leftarrow fact(u-1,w),\ times(u,w,v)$$

resolve them by invoking the *fact* procedure in response to the call to *fact* in the first one. This produces :-

$$table(x,z) \leftarrow fact(z-1,v''),\ times(z,v'',v'),\ enter(z,v',x',x),\ table(x',z-1)$$

after a little renaming of variables. The process of resolving them in this way contributes towards a compile-time symbolic solution of the *table* procedure's call to *fact*. The new procedure above requests the factorial of $(z-1)$ and so still gives rise to quadratic behaviour if substituted for the parent *table* procedure. Suppose instead, however, that this factorial is already accessible from the computation of the table x'; a procedure for accessing that factorial is trivially implied by the *table* specification, by virtue of the general fact that $A \leftrightarrow (B \leftrightarrow C)$ implies $C \leftarrow B,A$. Here the implied procedure is just, after some renaming :-

$$fact(z-1,v'') \leftarrow entry(z-1,v'',x'),\ table(x',z-1)$$

Now invoke this in response to the call $fact(z-1,v'')$ in the new *table* procedure to give :-

$table(x,z) \leftarrow table(x',z\text{-}1),\ entry(z\text{-}1,v'',x'),\ times(z,v'',v'),$
$enter(z,v',x',x)$

Observe that this contains no call to *fact*; thus the *fact* procedures are now computationally superfluous, playing no role in the new algorithm. If the same data structure representation is chosen as before, then the complete procedure set for the algorithm (which employs the same *table* basis as before) is :-

$table(e(0,1):\emptyset,\ 0) \leftarrow$
$table(e(z,v'):e(z\text{-}1,v''):x'',\ z) \leftarrow table(e(z\text{-}1,v''):x'',\ z\text{-}1),$
$times(z,v'',v')$

With Prolog-like control these execute just z multiplications in response to a goal $\leftarrow table(x,z)$ when $z>0$. However they have to be executed recursively, and so are not necessarily more efficient in practice. Essentially they behave rather like the result of executing bottom-up those procedures which comprise the former algorithm. Top-down execution of the present procedures for the goal $\leftarrow table(x,2)$ is depicted below in order to clarify their behaviour :-

$\leftarrow table(x,2)$	
$\leftarrow table(e(1,v''):x'',1),\ times(2,v'',v')$	$x:=e(2,v'):e(1,v''):x''$
$\leftarrow table(e(0,\hat{v}''):\hat{x}'',0),\ times(1,\hat{v}'',v''),$	
$times(2,v'',v')$	$x'':=e(0,v''):\hat{x}''$
$\leftarrow times(1,1,v''),\ times(2,v'',v')$	$\hat{x}'':=\emptyset\ ,\ \hat{v}'':=1$
$\leftarrow times(2,1,v')$	$v'':=1$
$\square$	$v':=2$

output : $x:=e(2,2):e(1,1):e(0,1):\emptyset$

The two anti-natural ordering algorithms are both inefficient in one way or another; the former executes too many multiplications, whilst the latter performs them in an order that demands recursive stacking of latent calls. In what follows we pursue the elimination of both of these defects.

Quadratic Natural Ordering Algorithm

The next three factorial table algorithms all compute tables in the natural order *(0,0!)*, ..., *(z,z!)*. Since *(0,0!)* is initially known in consequence of the specification set trivially implying :-

$$fact(0,1) \leftarrow$$

it can be summoned at the beginning of some computation and then used as the basis for generating higher entries. The previous algorithms had no immediate knowledge of any entries except *(0,0!)* either, but could not summon it immediately upon the start of computation; instead its role was deferred until completion of other recursive invocations initiated from requests for the higher entries.

A preliminary natural ordering algorithm can be obtained by admitting to *S* a new specification for a 3-place predicate *table**. Essentially this defines a partial table $\{(w,w!), \ldots, (z,z')\}$ as follows :-

$$table^*(x,w,z) \leftrightarrow (\forall uv)(entry(u,v,x) \leftrightarrow w \leqslant u \leqslant z,\ fact(u,v))$$

Choosing the instantiation $w:=0$ obviously makes *table(x,z)* and *table*(x,0,z)* *S*-equivalent, so that a call to the former can be investigated by a call to the latter.

It is very easy to pursue derivations for *table** which are closely analogous to those already seen for *table*. By using the consequences of the *enter* and *unit-table* specifications to reveal the construction of partial tables we obtain :-

$$table^*(x,z,z) \leftarrow fact(z,v'),\ unit\text{-}table(x,z,v')$$
$$table^*(x,w,z) \leftarrow w<z,\ fact(w,v),\ table^*(x',w+1,z),\ enter(w,v,x',x)$$

as the complete procedure set for *table** serving the initiating *table* procedure :-

$$table(x,z) \leftarrow table^*(x,0,z)$$

Using the term representation and macroprocessing, the resulting *table** procedures are iteratively executable :-

$$table^*(e(z,v'):\emptyset,\ z,z) \leftarrow fact(z,v')$$
$$table^*(e(w,v):x',\ w,z) \leftarrow w<z,\ fact(w,v),\ table^*(x',w+1,z)$$

A complete program body for solving a call to *table* will also clearly have to contain a suitable procedure set for solving the calls to *fact*. With top-down control, this algorithm must execute $\frac{1}{2}(z)(z+1)$

multiplications in the course of computing a table with entries up to *(z,z!)*, just like the earlier quadratic algorithm. So now we attempt a refinement as before with the aim of obtaining a linear behaviour instead.

Bi-linear Natural Ordering Algorithm

Once again we pursue a symbolic execution of the calls to *fact*, using a resolution step to eliminate those calls from the quadratic procedure set. Considering the recursive *table** procedure first, the call *fact(w,v)* is required to be replaced by a look-up of the next factorial in question from the table *x'*. This factorial will clearly be *(w+1)!* by virtue of the sentence implied by *S* (similar to that which was exploited in the earlier refinement), which after renaming is :-

$$fact(w+1,v'') \leftarrow entry(w+1,v'',x'),\ table^*(x',w+1,z)$$

This means that a procedure is required which solves *fact(w,v)* by solving *fact(w+1,v")*. Now it is trivial to show that the specification given for *fact* implies :-

$$(\exists v)(fact(w,v),\ times(w+1,v,v'')) \leftarrow fact(w+1,v''),\ 0 \leqslant w$$

from which it is possible to infer the universally quantified sentence :-

$$fact(w,v),\ times(w+1,v,v'') \leftarrow fact(w+1,v''),\ 0 \leqslant w$$

because of the assumptions introduced earlier that *v* is uniquely determined in the consequent formula above for any given choice of *w* and *v"*. Then since, generally, $(A,B) \leftarrow C$ implies $A \leftarrow B,C$ the desired procedure for *fact* is immediately obtained :-

$$fact(w,v) \leftarrow fact(w+1,v''),\ times(w+1,v,v''),\ 0 \leqslant w$$

and resolved with the recursive *table** procedure above to produce a new procedure having no explicit call to *fact*. There is no similar way of eliminating the call to *fact* in the *table** basis, and it is important to understand why this is so: the reason is that if it were eliminated, the resulting procedure set would not have any intrinsic knowledge about any particular factorials; this contrasts with the linear procedure set for *table* which, despite having no calls to *fact* nevertheless has the entry *(0,1)* embedded in the first argument of the basis. In the present case there is no way of

assimilating the immediate knowledge of that entry in the basis, whose task is not to compute *0!* but rather any arbitrary *z!* where *(z,z!)* is the last entry of the desired table. Therefore the best we can achieve in modifying the *table** procedures is the set :-

table(e(z,v'):∅, z,z) ← fact(z,v')*

table(e(w,v):e(w+1,v"):x",w,z) ← 0≤w, w<z, times(w+1,v,v"),*
table(e(w+1,v"):x",w+1,z)*

[together with procedures for fact]

Executed top-down to solve a goal like ← *table*(e(0,1):x',0,z)* for some given input instance of *z*, these can be executed iteratively to generate entries in the natural order. Note that a goal of that kind effectively injects an initial factorial entry *(0,1)* into the computation from its first argument; if the first argument were simply an output variable instead, as in ← *table*(x,z)*, then the calls preceding the recursive procedure's call to *table** could not be activated deterministically, disallowing an iterative computation. We call the algorithm 'bi-linear' here to reflect the fact that the total number of multiplications which it executes is *2z* for the goal ← *table*(e(0,1):x',0,z)* where *z>0*. This is unsatisfactory but certainly an improvement upon *½(z)(z+1)*.

Note that if the goal is set up as above with a *correct* entry in its first argument, the calls *fact(z,v')* and *0≤w* can be deleted from the modified *table** procedure set above and still give a successful computation but with only *z* multiplications. This is a perfectly satisfactory computation, *but the table* procedures no longer conform to the specification set ;* that is, it is no longer possible to show that they are true theorems about the various relations as specified by *S*; they execute 'correctly' only if given a goal which provides a genuine entry in its first argument. The presence of those two calls in the truly correct procedures can be interpreted as a constraint which checks that any entry injected by the goal is (i) in a valid range, that is, higher than the entry for *0!* or equal to it, and (ii) a pair *(u,v)* such that *v* really is the factorial of *u* : the latter being checked indirectly by using whatever entry is given in order to compute some *v'* as the potential factorial of *z*, and then explicitly checking that this value of *v'* really is the factorial of *z* through the agency of the call to *fact* in the basis. It is the latter check on the goal's integrity which gives rise to the extra *z* multiplications in the algorithm.

Linear Natural Ordering Algorithm

The last algorithm considered is the best of those examined so far. The failing which has still to be dealt with is that it requires the invoking call to supply a known initial entry (in order to obtain deterministic iterative behaviour), and that it is then burdened by an internal check applied to that initial entry. The final algorithm presented here eliminates these irritations and thus gives impeccable behaviour. The price of this is the need to resort to slightly unobvious intuitions in order to find the right specification. It so happens that a further predicate has to be specified in order to provide procedures for a 4-place relation *table***.

In contemplating the failings of the *table** program just examined, it is possible to perceive that its final call to *fact* would be unnecessary if it had been executed inherently upon the condition that the injected entry *(w,v)* satisfied *w! = v*. This is the intuition which underlies the *table*** specification below, which is also admitted to *S* :-

$$table^{**}(x,w,v,z) \leftrightarrow (table^{*}(x,w,z) \leftarrow fact(w,v))$$

It is easy to see that this trivially implies :-

$$table^{*}(x,w,z) \leftarrow fact(w,v),\ table^{**}(x,w,v,z)$$

Choosing the case *w:=0* and invoking the *fact* basis to eliminate the call to *fact* in this sentence, an initiating procedure for *table* is readily obtained :-

$$table(x,z) \leftarrow table^{**}(x,0,1,z)$$

so that an entry *(0,1)* known to be correct is inherently built in to the new procedure set. Once again, the axioms of constructibility of tables assembled already in *S* are sufficient to permit derivations for both a recursion and a basis on *table***. In particular it is possible to take a short cut in the derivation of the basis by observing that the *table** basis :-

$$table^{*}(e(z,v'):\emptyset,\ z,z) \leftarrow fact(z,v')$$

matches the *table*** definiens, thereby immediately providing a *table*** basis :-

$$table^{**}(e(z,v):\emptyset,\ z,v,z) \leftarrow$$

The recursive *table*** which completes the necessary procedure set is derivable without difficulty just like the recursive *table* procedure :-

$table^{**}(e(w,v):x',\ w,v,z) \leftarrow 0 \leqslant w,\ w<z,\ times(w+1,v,v''),$
$table^{**}(x',w+1,v'',z)$

Executed with Prolog-like control, this set of three procedures gives z multiplications in order to solve the goal $\leftarrow table(x,z)$. Entries are computed in the natural order and no initial entry need be injected through the goal. The computation is also deterministic and iterative. It will probably be helpful to present an example of a computation using them to compute the table for the case $z:=2$. This computation is depicted below with the $\leqslant$ checks omitted for clarity, although it is, of course, assumed that they have been properly executed.

$\leftarrow table(x,2)$	
$\leftarrow times(1,1,v''),\ table^{**}(x',1,v'',2)$	$x:=e(0,1):x'$
$\leftarrow table^{**}(x',1,1,2)$	$v'':=1$
$\leftarrow times(2,2,\hat{v}''),\ table^{**}(x'',2,\hat{v}'',2)$	$x':=e(1,1):x''$
$\leftarrow table^{**}(x'',2,2,2)$	$\hat{v}'':=2$
$\square$	$x'':=e(2,2):\emptyset$

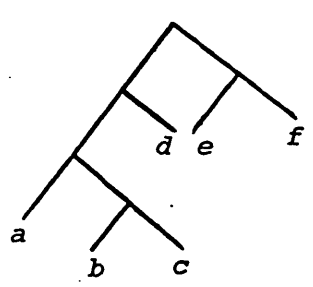
output : $x := e(0,1):e(1,1):e(2,2):\emptyset$

6.4 : COMPARISON OF TREE FRONTIERS

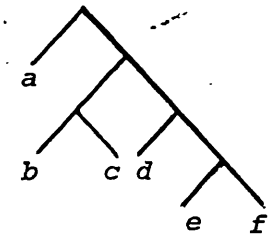
Specifying the Problem

This example deals with the well-known problem of determining whether or not two given trees have identical frontiers. For simplicity of presentation, the input data is here assumed to be restricted to *binary* trees, but this does not reduce the generality of the algorithm employed. The most interesting aspect of that algorithm is a special data structure transformation applied to each of the two trees of interest which assists the task of accessing and comparing the members of their frontiers. This transformation can be assimilated into the logical derivation of the frontier-accessing procedures by proving a simple theorem about the associative constructibility of the trees' representations.

To specify the problem formally it is necessary to define the data structures involved. A binary tree is representable by a pair (x_1,x_2) of which each component is either a binary tree or a labelled tip ; in any particular tree the tips are labelled distinctly. The frontier of a binary tree (x_1,x_2) is the result of appending the frontier of x_2 to the frontier of x_1 ; the frontier of a labelled tip is the unit list whose member is that tip's label. Thus each binary tree has a unique frontier consisting of a list of distinct tip labels. However, associated with any given frontier there exist finitely many binary trees possessing that frontier; each tree corresponds to one way of constructing the frontier by appending its constituent sublists. As an example, two trees shown below are selected from *42* distinct binary trees which all possess the frontier *(a,b,c,d,e,f)*.



two trees with same frontier.



This pair might constitute the data for our problem, that is, to show that they do indeed have the same frontier.

To construct a logical specification it is convenient to introduce the predicate *label(u,i,x)* to express the fact that u is the i^{th} label in the frontier of binary tree x. Furthermore let the predicate *same-frontier(x,y)* hold when trees x and y have the same frontier. The latter predicate names the primary relation of interest, and can be expressed in terms of the *label* predicate as follows :-

$$same\text{-}frontier(x,y) \leftrightarrow (\forall ui)(label(u,i,x) \leftrightarrow label(u,i,y))$$

Thus, given two trees and the means of determining their frontier labels, we require a procedure set capable of solving a call to *same-frontier*. This is the object of the ensuing derivations. An approach to the problem which is rather different from that given here may be found in the paper by Burstall and Darlington (10).

The Conventional Algorithm

The most naive algorithm for the problem is that which compares the i^{th} labels in the given trees x and y choosing $i = 1, 2, \ldots$, etc. in sequence. This is the algorithm which is considered here; the central problem which it poses for the programmer is how to access the i^{th} label in the frontier of a given binary tree having otherwise arbitrary topology. In fact it turns out to be unnecessary to devise procedures which explicitly seek a particular i^{th} label, as will be shown shortly.

Suppose that the frontier of x is some list $(u_1, \ldots, u_m)$ whilst that of y is some list $(y_1, \ldots, y_n)$. If u_1 and v_1 are successfully matched, it remains to compare the sublists $(u_2, \ldots, u_m)$ and $(v_2, \ldots, v_n)$. In devising procedures for the logic component of the algorithm which makes these comparisons, it is possible - and indeed advantageous - to avoid explicit reference to (and hence computation of) these sublists by postulating the existence of trees x' and y' whose frontiers are respectively $(u_2, \ldots, u_m)$ and $(v_2, \ldots, v_n)$. We shall see that to solve the problem of matching the labels it is sufficient to match representations of these trees associated with sublists of the trees' frontiers, rather than having to match explicit list representations of those frontiers. With this in mind, then, admit to the specification set a sentence :-

$$split\text{-}frontier(u',x',x) \leftrightarrow (\forall ui)(label(u,i,x) \leftrightarrow label(u,i-1,x') \vee (u=u',i=1))$$

The predicate *split-frontier(u',x',x)* holds when the frontier of tree *x* is the result of appending the frontier of tree *x'* to the unit list whose member is the tip label *u'*; it provides a means of explicating the construction of tree frontiers. Now there are also trees whose frontiers are not expressible in this manner, namely trees consisting of single tip nodes; their relationship to their frontiers is expressible using the predicate *unit-tree(x,u')* which holds when the frontier of tree *x* is the unit list whose member is the tip label *u'*. This predicate is specified in *S* by :-

$$unit\text{-}tree(x,u') \leftrightarrow (\forall ui)(label(u,i,x) \leftrightarrow u=u',\ i=1)$$

From the experience of previous examples presented here, it should be clear that the two construction axioms for frontiers just described can be used as a source of *S-conditional-equivalence substitutions* for *label* predicates in a derivation for *same-frontier* which begins :-

$$\leftarrow same\text{-}frontier(x,y)$$
$$\leftarrow (\forall ui)(label(u,i,x) \leftrightarrow label(u,i,y))$$

By exploiting the ways of constructing frontiers expressed in the specification set, this derivation branches in order to deal with unit trees on one branch and more general trees on the other. Some trivial goal substitutions and simplifications then lead to the complete procedure set for *same-frontier* :-

$$same\text{-}frontier(x,y) \leftarrow unit\text{-}tree(x,u'),\ unit\text{-}tree(y,u')$$
$$same\text{-}frontier(x,y) \leftarrow split\text{-}frontier(u',x',x),\ split\text{-}frontier(u',y',y),\ same\text{-}frontier(x',y')$$

Note that there is no need to devise a procedure catering for empty trees because it is assumed that there is no such kind of tree in this particular formulation of the problem.

It now remains to synthesize procedure sets for *unit-tree* and *split-frontier*. In other examples we have met similar circumstances where it was required to compose procedure sets for almost analogous relations such as *unit-list* and *append**, and it was seen that by choosing

particular data structure representations the calls to those data-accessing procedures could be trivially eliminated by macro-processing. In the present case, however, a concrete tree representation is chosen which allows the calls to *unit-tree* to be eliminated, whilst those to *split-frontier* are not; instead they invoke procedures for *split-frontier* which provide a rather subtle means of computed access to the first label u' in the concrete representation of any tree z and simultaneously construct the tree z' whose frontier is the result of deleting u' from the frontier of z. The behaviour of the procedures for *split-frontier* is the central feature of the whole algorithm.

Suppose, then, that the trees of interest are represented concretely by terms using constructors t and λ, such that the term $t(z_1,z_2)$ denotes a binary tree with left-tree z_1 and right-tree z_2, whilst the term $\lambda(u)$ denotes a unit tree whose tip is labelled u. Moreover, introduce a predicate $numtips(z,j)$ to express the fact that a tree z has j tip labels in its frontier. Then the following sentences can be added to S in order to specify the meaning of *label* for the chosen representation :-

$$label(u,i,\lambda(u')) \leftrightarrow (u=u',\ i=1)$$

$$label(u,i,t(z_1,z_2)) \leftrightarrow (\exists i_1)(numtips(z_1,i_1),\ (label(u,i,z_1) \vee label(u,i-i_1,z_2)))$$

together with three axioms constraining the well-formedness of trees analogous to the list axioms *A1-A3* :-

$$(\exists j)numtips(z,j) \leftarrow$$

$$numtips(z,j) \leftrightarrow (\forall i)(1 \leqslant i \leqslant j \leftrightarrow (\exists u)label(u,i,z))$$

$$(\forall u)(label(u,i,z) \leftrightarrow u=\hat{u}) \leftarrow label(\hat{u},i,z)$$

The resources now established in S provide for the derivation of a rather subtle property of the *split-frontier* relation which we shall employ as a *split-frontier* procedure. In pursuing the derivation goal :-

$$\leftarrow split\text{-}frontier(u',x',x)$$

two cases are possible for the structure of the tree x if x is decomposable : either it has the form $t(t(x_1,x_2),x_3)$ or else it has the form $t(\lambda(\hat{u}'),\hat{x}')$. This just represents a case analysis on the left-tree.

In dealing with the former case by pursuing the goal :-

$\leftarrow$ *split-frontier*$(u',x',t(t(x_1,x_2),x_3))$

a lemma will be invoked which underlies the logic of the target algorithm. The usefulness of this lemma is not easy to perceive by merely considering the derivation goal and depends upon some considerable inspiration. Here it is derived from *S* beforehand by somewhat bottom-up inferences as follows. Recalling the analogous treatment of the list reversal problem examined in Section 6.1, the uniqueness of a tree's tip count can be proved from the axioms of well-formedness and then exploited so as to re-organize the specification of *label*$(u,i,t(z_1,z_2))$ as follows :-

$$(\forall ui)(label(u,i,t(z_1,z_2)) \leftrightarrow label(u,i,z_1) \vee label(u,i-i_1,z_2)) \leftarrow numtips(z_1,i_1)$$

This will now be used as a source of *S-conditional-equivalence substitutions* applied to itself for the instances $z_1:=t(x_1,x_2)$, $z_2:=x_3$ chosen above in the derivation goal. For ease of presentation, let *F* abbreviate the formula *label*$(u,i,t(t(x_1,x_2),x_3))$; then the desired lemma is proved as follows :-

$$(\forall ui)(\ F \leftrightarrow label(u,i,t(x_1,x_2)) \vee label(u,i-i_{12},x_3)) \leftarrow numtips(t(x_1,x_2),i_{12})$$

$$\vdash_S (\forall ui)(\ F \leftrightarrow label(u,i,x_1) \vee label(u,i-i_1,x_2) \vee label(u,i-i_{12},x_3)) \leftarrow numtips(t(x_1,x_2),i_{12}),\ numtips(x_1,i_1)$$

$$\vdash_S (\forall ui)(\ F \leftrightarrow label(u,i,x_1) \vee label(u,i-i_1,t(x_2,x_3))) \leftarrow numtips(t(x_1,x_2),i_{12}),\ numtips(x_1,i_1),\ numtips(x_2,i_2)$$

$$\vdash_S (\forall ui)(\ F \leftrightarrow label(u,i,t(x_1,t(x_2,x_3)))) \leftarrow numtips(t(x_1,x_2),i_1+i_2),\ numtips(x_1,i_1),\ numtips(x_2,i_2) \quad [i_{12} := i_1+i_2]$$

This is the desired lemma. Informally, it says that if *u* is the i^{th} label in the frontier of $t(t(x_1,x_2),x_3)$ then it is equivalently the i^{th}

label in the frontier of $t(x_1,t(x_2,x_3))$. The *numtips* antecedents can be deleted by proving the existences of the tip counts from the axioms already in S; this establishes that S implies the sentence :-

$$\underline{\textit{lemma}}: \quad label(u,i,t(t(x_1,x_2),x_3)) \leftrightarrow label(u,i,t(x_1,t(x_2,x_3)))$$

The derivation of the general procedure for *split-frontier* is now trivial, using the lemma to make an *S-equivalence substitution* :-

$\leftarrow split\text{-}frontier(u',x',t(t(x_1,x_2),x_3))$
$\leftarrow (\forall ui)(label(u,i,t(t(x_1,x_2),x_3)) \leftrightarrow label(u,i-1,x') \vee (u=u',i=1))$
$\leftarrow (\forall ui)(label(u,i,t(x_1,t(x_2,x_3))) \leftrightarrow label(u,i-1,x') \vee (u=u',i=1))$
$\leftarrow split\text{-}frontier(u',x',t(x_1,t(x_2,x_3)))$

from which is inferred the procedure :-

$$split\text{-}frontier(u',x',t(t(x_1,x_2),x_3)) \leftarrow split\text{-}frontier\ (u',x',t(x_1,t(x_2,x_3)))$$

When the derivation pursues the alternative case for the structure of the tree, the call in the goal is simplified by using the properties of unit trees already established in S, as follows :-

$\leftarrow split\text{-}frontier(u',x',t(\lambda(\hat{u}'),\hat{x}'))$
$\leftarrow (\forall ui)(label(u,i,t(\lambda(\hat{u}'),\hat{x}')) \leftrightarrow label(u,i-1,x') \vee (u=u',i=1))$

[modus tollens]

$\leftarrow (\forall ui)((label(u,i,\lambda(\hat{u}')) \vee label(u,i-\hat{i},\hat{x}')) \leftrightarrow$
$(label(u,i-1,x') \vee (u=u',i=1))),$
$numtips(\lambda(\hat{u}'),\hat{i})$

[S-cond.-equiv. substitution, invoking label specification]

$\leftarrow (\forall ui)(((u=\hat{u}',i=1) \vee label(u,i-1,\hat{x}')) \leftrightarrow$
$(label(u,i-1,x') \vee (u=u',i=1)))$

[using property of unit tree $\lambda(\hat{u}')$ in S that it has just one label, thus inducing $\hat{i}$:=1 and solving last call]

□ *[simplifying by instantiation to delete call : $\hat{x}'$:=x', $\hat{u}'$:=u']*

from which is inferred the *split-frontier* basis procedure :-

$$split\text{-}frontier(u',x',t(\lambda(u'),x')) \leftarrow$$

This completes the synthesis for *split-frontier*. There remains the matter of dealing with the calls to *unit-tree* in the basis for *same-frontier*. Now the specifications already given for *unit-tree* and for the meaning of *label* for trees represented by $\lambda(u)$ jointly imply the assertion :-

$$unit\text{-}tree(\lambda(u),\ u) \leftarrow$$

so that the calls to *unit-tree* can be macro-processed out to leave the basis :-

$$same\text{-}frontier(\lambda(u'),\ \lambda(u')) \leftarrow$$

This basis is perfectly adequate for terminating the successive decomposition of frontiers by the *split-frontier* procedures. However, the behaviour of the algorithm is such that it is possible for the recursive *same-frontier* procedure to generate identical sub-trees x' and y' whose frontiers necessarily coincide, so that computation could then be terminated immediately with the frontiers of the given trees successfully matched. Thus a considerable gain in efficiency is possible by generalizing the basis above to :-

$$same\text{-}frontier(x,x) \leftarrow$$

Clearly this is trivially implied by the *same-frontier* specification through the instantiation $y:=x$, and also implies the basis above which was derived specifically for dealing with unit trees. When computation is terminated by this basis in the case where x' and y' are not unit trees, the matching of them is accomplished through this single procedure invocation in consequence of the unification mechanism. The present example bears similarities to the example discussed in Chapter 3 dealing with programs for investigating the list equality relation. Here then is the final procedure set for the present problem :-

$$same\text{-}frontier(x,x) \leftarrow$$

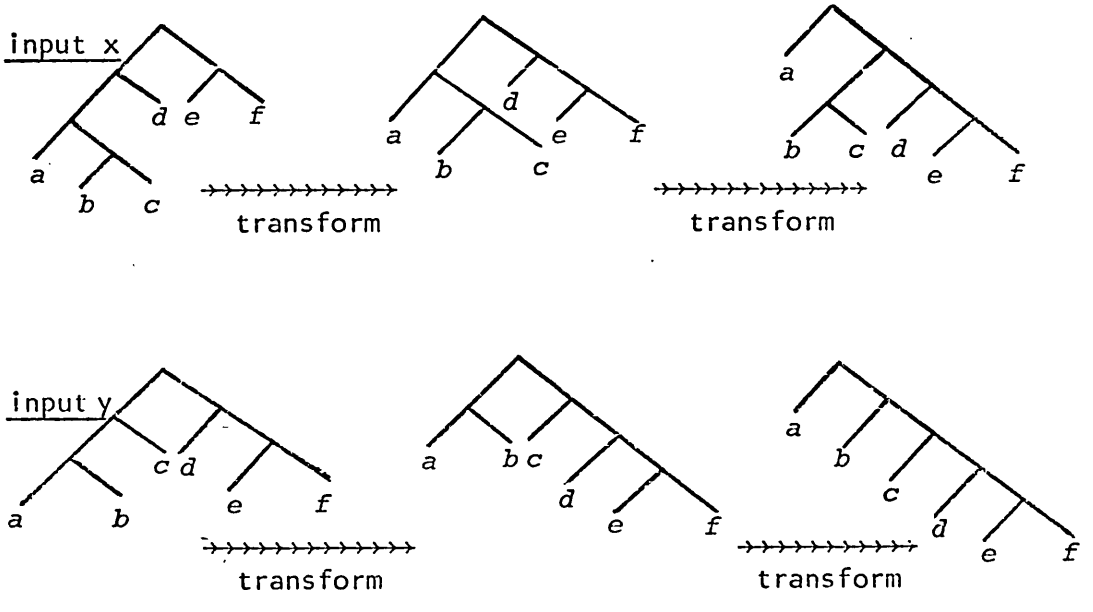
$$same\text{-}frontier(x,y) \leftarrow split\text{-}frontier(u',x',x),\ split\text{-}frontier(u',y',y),\ same\text{-}frontier(x',y')$$

$$split\text{-}frontier(u',x',\ t(\lambda(u'),x')) \leftarrow$$

$$split\text{-}frontier(u',x',\ t(t(x_1,x_2),x_3)) \leftarrow split\text{-}frontier(u',x',t(x_1,t(x_2,x_3)))$$

With Prolog-like control these procedures generate the rather charming algorithm which successively transforms the given trees in order to compare their first frontier labels. If these do not match then termination is immediate; if they do match, the transformation which made them accessible will also have generated two more trees respectively associated with the reduced frontiers, and then the computation proceeds to transform these in a similar manner. The general behaviour of the algorithm can be seen as a series of label comparisons before each of which there is a series of tree transformations which accesses the labels to be next compared. The logic of this transformation process is summed up in the recursive procedure for *split-frontier* which makes the first label in the tree $t(t(x_1,x_2),x_3)$ more accessible by seeking it in the tree $t(x_1,t(x_2,x_3))$ *[thus reducing its depth in the tree being searched]* in the knowledge that this transformation preserves the frontier.

A slice selected from a simple computation is depicted below in order to indicate the algorithm's strategy.



The input trees x and y above are transformed by repeated recursions on the *split-frontier* procedure until both calls to *split-frontier* in the *same-frontier* procedure return identical instances of the first label u', which is a in the example above. The reduced trees x' and y' having the frontier (b,c,d,e,f) are then compared using similar transformations until b becomes accessible. Here, the tree x, for example, is represented by the term :-

$$t(t(t(\lambda(a),t(\lambda(b),\lambda(c))),\lambda(d)),t(\lambda(e),\lambda(f)))$$

6.5 : SUMMATION OF MATRIX TRANSVERSES

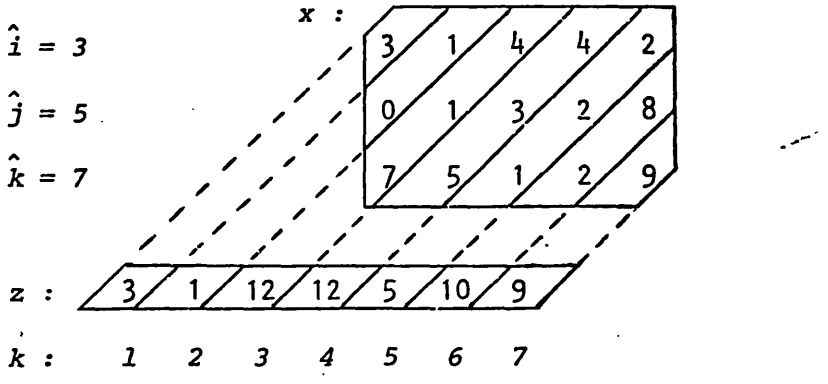
Specifying the Problem

This example considers the problem of computing the sums of the elements on the transverse diagonals of a given matrix and storing those sums in a list. There are two quite different approaches to this task which differ in efficiency : firstly, one can compute the members in that list sequentially, so that to compute any one of them it is necessary to use some accessing protocol which finds just those elements in the matrix which occur in the transverse summed by that particular list member; secondly, one can access the elements from the matrix in any convenient way and, for any one of them, decide which transverse it occurs in and so add it to a cumulative sum in the list associated with that transverse. In the former case complete sums are generated serially, whereas in the latter case those sums are built up in quasi-parallel. The difference in efficiency is determined by the computations necessary for associating particular members of the list with particular elements in the matrix. In the first algorithm some k is known as the index of the transverse whose sum is to be computed, whence it is then necessary to generate just those pairs (i,j) of coordinates in the k^{th} transverse of the matrix; this involves some untidy counters and associated bounds. In the second algorithm some pair (i,j) is given which selects the next element of the matrix to be added to some k^{th} transverse's sum; the computation of the relevant k is trivial - it is just $k = i+j-1$.

In both algorithms it is favourable to efficiency to arrange an essentially bottom-up generation of the cumulative sums, rather than pursuing top-down recursive evaluations. For this reason, both logic programs shown here - which are intended for top-down interpreters - resort to the programming styles which simulate bottom-up behaviour. Both of them use procedures which maintain explicit pointers in their argument structure which govern the access to the matrix, and so their derivations require the kind of techniques for arranging this which have already appeared in other examples. A synthesis is not given here for the logic representation of the first algorithm because this would not comprise particularly difficult or interesting derivations. However, the greater part of the ideas needed for synthesizing the logic

component of the second algorithm is presented in some detail to show, for the first time in the thesis, a non-trivial derivation which exploits Kowalski's -arity-doubling technique for specifying procedures which simulate bottom-up behaviour. Furthermore, whereas this has been employed by others with the simple aim of actually using those procedures for computational purposes [for example, Kowalski's use (51) of the *go** procedures and Clark's use (12) of the 4-place factorial procedures], here we do not employ it exactly in that way, but rather use it to introduce a specification of a relation which only plays an intermediate, but nonetheless important, role as an axiom invoked during a derivation. Whereas the typical use of the technique just reverses the direction of a serial computation, its use here results in the transformation of a serial computation to a quasi-parallel one.

It is now appropriate to introduce some notation in order to specify the problem in greater detail. Let x be the given matrix having $\hat{i}$ rows and $\hat{j}$ columns, where $\hat{i} \geq 1$ and $\hat{j} \geq 1$; when this is so, the predicate $size(x,\hat{i},\hat{j})$ is satisfied. Then associated with x there exist $\hat{k}$ distinct 'transverses' which can be labelled $1, \ldots, \hat{k}$. The k^{th} transverse is that substructure of the matrix which consists of just those elements $x[i,j]$ satisfying $k = i+j-1$; this definition determines the partitioning of the matrix elements into $\hat{k}$ disjoint non-empty transverses. Associated with x is a list z such that any v is a k^{th} member of z if and only if it is the sum of the elements in the k^{th} transverse of x; when this is so, the predicate $sumlist(x,z)$ is satisfied. The picture below should help to clarify these ideas and notations.



The formal specification of *sumlist* is given by the sentence :-

$$sumlist(x,z) \leftrightarrow (\forall vk)(item(v,k,z) \leftrightarrow transum(v,k,x))$$

where the predicate *transum(v,k,x)* holds when v is the sum of the elements in the k^{th} transverse of x. The look-up of elements whose coordinates are known is implemented by calls to procedures for a relation *elem(u,i,j,x)* which holds when $u = x[i,j]$. For instance, the matrix might be represented by an array of *elem* assertions, thereby allowing direct access. Whichever arrangement is chosen in practice is not important to the analysis given here, which is concerned only with the *order* in which calls to *elem* are executed; thus *elem* is left as a primitive in the specification set. A few more relations will be specified in S when the need arises.

The Serial Summation Algorithm

The serial algorithm is that firstly described in the previous introduction to the problem; it serially computes the list members $z[k]$ for $k = 1, \ldots, \hat{k}$ in that order. Each time some new k is chosen, the algorithm has to determine the elements on the k^{th} transverse. Probably the simplest way to achieve this is to compute a 'start-address' *(i,j)* for the transverse at which its first element can be found; more precisely, we can arrange that its i component is the least row coordinate on the transverse (that is, the one nearest the top of the diagram). This address is computable by the expression :-

$$(i,j) := \underline{if}\ k \leqslant \hat{j}\ \underline{then}\ (1,k)\ \underline{else}\ (k-\hat{j}+1,\hat{j})$$

In order for the algorithm to iterate through the selection of elements from the transverse, it must also know how to recognize the last one selected. The most efficient arrangement here is to compute in advance the number of elements on the transverse and then count the elements selected. If k' represents this count, it is computable using :-

$$k' := \underline{if}\ 1 \leqslant k < \hat{i}\ \underline{then}\ k\ \underline{else}\ \underline{if}\ \hat{i} \leqslant k \leqslant \hat{j}\ \underline{then}\ \hat{i}\ \underline{else}\ \hat{k}-k+1$$

Once these preliminaries have been accomplished, the required elements on the k^{th} transverse can be accessed successively (proceeding 'down' the transverse) by incremental address modification until all k' of them have been accessed; meanwhile, of course, they are being added to a cumulative sum whose final state will be the k^{th} member assigned to z. It will be useful to show the conventional representation of this arrangement before discussing a possible logic representation. Thus a simple Algol-like rendering of the serial algorithm is :-

```
begin for k := 1 to î+ĵ-1 do        [ iteration through transverses
      begin                           in succession]
            compute(î,ĵ,k,i,j,k'); [ get start-address (i,j), count k']
            k" := 1 ;
            v  := x[i,j] ;          [ initializing transverse sum]
            while k"<k' do          [ iteration through elements on
                  begin               k^th transverse]
                        i := i + 1 ;
                        j := j - 1 ;
                       k" := k"+ 1 ;
                        v := v + x[i,j]  [adding to transverse sum]
                  end
            z[k]:= v                      [assigning transverse sum
      end                                  to k^th member of output list]
end
```

The algorithm above can be represented without much difficulty as a Prolog-like execution of a logic program using procedures for the relations specified earlier. One possible rendering now follows, in which the predicate *compute*$(\hat{i},\hat{j},k,i,j,k')$ holds when the k^{th} transverse of an $\hat{i}$ x $\hat{j}$ matrix has k' elements and a start-address (i,j). It may be assumed that the call to *compute* just evaluates i, j and k as specified by the rule given previously. The predicates *sumlist** and *transum** are just variants of *sumlist* and *transum* which explicate the indices governing the iterations, and are like the predicates *ord** and *palin*** seen in Chapter 3.

```
sumlist(x,z) ← size(x,î,ĵ), k̂ = î + ĵ - 1, sumlist*(x,1,k̂,z)

sumlist*(x,k̂,k̂,v.nil) ← transum(v,k̂,x)
sumlist*(x,k,k̂,v.z')  ← k<k̂, transum(v,k,x)

transum(v,k,x) ← size(x,î,ĵ), compute(î,ĵ,k,i,j,k'),
                              transum*(v,i,j,1,k',x)

transum*(u,i,j,k',k',x) ← elem(u,i,j,x)
transum*(u+v',i,j,k",k',x) ← k"<k', elem(u,i,j,x),
                             transum*(v',i+1,j-1,k"+1,k',x)
```

These procedures are a more-or-less direct transcription of the conventional program above. The predicate *sumlist*$(x,k,\hat{k},z")$ holds

when z'' is the list of the sums of the k^{th} up to the $\hat{k}^{th}$ transverses of x; the predicate *transum*(v",i,j,k",k',x)* means that v'' is the sum of the k''^{th} up to the k'^{th} elements on the $(i+j-1)^{th}$ transverse of x. The procedures behave quite acceptably in response to a goal ← *sumlist(x,z)* , successively binding new transverse sums to the output argument z. Their inefficiency in having to call the *compute* procedure in order to select the elements on each transverse is no worse than the conventional program's behaviour. This particular computational burden is, nevertheless, a nuisance, and in the next discussion we eliminate it altogether. The result is not only greater efficiency but also a significant shortening and clarification in the texts of both the conventional program and the logic program.

The Quasi-parallel Summation Algorithm

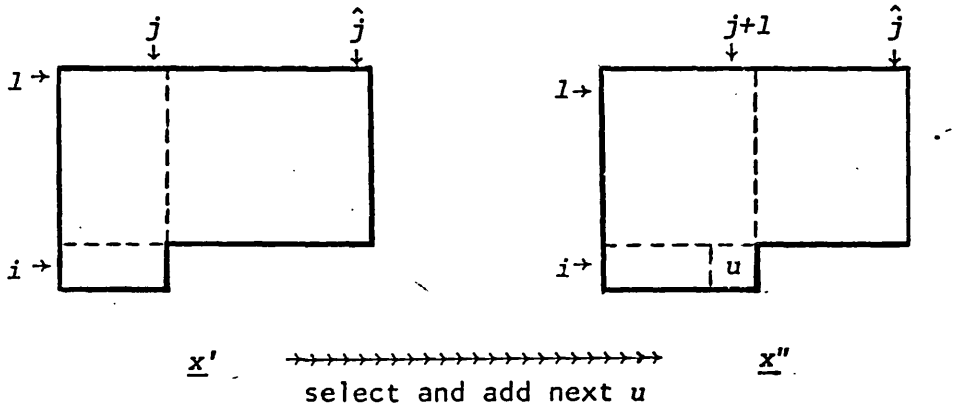
Since every element in the matrix contributes to some transverse, there exists an algorithm for the problem at hand which exhaustively accesses elements from x and adds each one to a cumulative sum of the elements in its associated transverse. The order in which the elements are selected from x is inconsequential to the fact that by the time they have all been selected and added to the appropriate sums, the final state of the output list will be correct. In general the sums in the list z are not computed successively (although this can still be arranged) but rather are built up in quasi-parallel. The conventional program below depicts the algorithm which uses the easiest method for selecting the elements, that is, by executing a column-selecting iteration within a row-selecting iteration, each in the natural order *1, 2, ... etc.*

```
begin for k := 1 to î+ĵ-1 do z[k] := 0 ;
      for i := 1 to î do
      for j := 1 to ĵ do
          begin
                k := i + j - 1 ;
                z[k] := z[k] + x[i,j]
          end
end
```

This is certainly much less clumsy than the program for the serial

algorithm in appearance and better behaved in execution. Now we shall show that the same improvements can be realized in the logic representation of this algorithm.

Since the order of selection of elements is of no logical consequence it is sensible to choose the natural orderings of row and column coordinates as the protocol for the logic program, just as in the above conventional program. For this purpose it is convenient to imagine a substructure x' of x which grows in a uniform way as computation proceeds under the control of natural row and column selection; at any particular instant the substructure is just that part of the matrix whose elements have so far been added to the appropriate cumulative sums in z. Let the term $s(x,i,j)$ represent the substructure x' as depicted below. The successor state in the computation will be either $s(x,i,j+1)$ (if $j<\hat{j}$) or else $s(x,i+1,1)$ (if $j=\hat{j}$), provided that x' is not the final state ($i=\hat{i}$, $j=\hat{j}$). This successor state is also shown below (named x'') revealing the newly selected element $u = x[i,j+1]$.



Considering the algorithm in a general way, imagine that the list z is constructed incrementally by the successive additions of elements to its various members. Initially z will consist of a list z^o consisting of $\hat{k}$ zeros; this is associated with the state $s(x,1,0)$ of the substructure x' of selected elements. More generally, each time some state x' is promoted to x'', the state of z will be promoted from z' to z''. The objective is to compute the final state $\hat{z}$ associated with $x' = s(x,\hat{i},\hat{j})$ from z^o.

The objective just outlined can be expressed in a preliminary way in terms of a new predicate *sumlist**(x,z,x',z')* which is interpreted to mean that *z* is the list of transverse sums of elements in the substructure *x* if *z'* is the list of transverse sums of elements in the substructure *x'*. Logically this is expressed :-

$$sumlist^{**}(x,z,x',z') \leftrightarrow (sumlist(x,z) \leftarrow sumlist(x',z'))$$

and so is exactly an instance of Kowalski's typical specification style for some anticipated *sumlist*** program whose top-down execution behaves like the bottom-up execution of some recursive program for *sumlist*. The algorithm which is of interest here is that which solves a call $sumlist^{**}(s(x,\hat{i},\hat{j}),\hat{z},s(x,1,0),z^{\circ})$ given some data structure representing the zero-filled list z° and, of course, access to *x*.

Whilst the *s*-notation is useful for descriptive purposes, it is not desirable that it should remain in the argument structure of calls in the eventual program, partly because it clutters the text, and partly because it places an overhead on run-time unification (unless it can be eliminated by some compile-time inferences). At this stage, then, we make a transformation of the principal predicate of interest by giving it the name $sumlist^{\dagger}$ and the specification :-

$$sumlist^{\dagger}(x,\hat{i},\hat{j},z,i,j,z') \leftrightarrow size(x,\hat{i},\hat{j}),\ sumlist^{**}(x,z,s(x,i,j),z')$$

Now the essential logic of the problem is obviously concerned with the relationship between successive states of the list *z* as *x'* progresses through successive states. Two axioms can be added to *S* which summarize some simple and (we hope) clearly correct facts concerning the relations of interest. In the first place, it should be clear that if *z"* is the succeeding state to *z'*, as expressed by a predicate *add(u,i,j,z',z")* holding when the addition of *u* to the $(i+j)^{th}$ member of *z'* produces *z"*, then all members of *z'* and *z"* will be correspondingly identical except for their $(i+j)^{th}$ members which will differ by *u* :-

$$(\forall vk)\,(item(v,k,z'') \leftrightarrow (item(v,k,z'),\ k\neq i+j) \vee (item(v-u,k,z'),\ k=i+j)) \leftarrow add(u,i,j,z',z'')$$

Likewise, if the condition *elem(u,i,j+1,x)* holds, it is easy to see that the transverse sums of the substructures *s(x,i,j)* and *s(x,i,j+1)* will be correspondingly identical except for their $(i+j)^{th}$ transverse sums which will differ by *u* :-

$$(\forall v k)\,(transum(v,k,x'') \leftrightarrow (transum(v,k,x'),\ k\neq i+j) \vee (transum(v-u,k,x'),\ k=i+j)) \leftarrow elem(u,i,j+1,x)$$

From these two axioms it is then easy to combine them (by conjunction), simplify the result and exploit the *sumlist* specification to arrive at what will prove to be a useful lemma; it just describes an obvious fact about the states of the data structures when a new element is selected and added :-

$$(sumlist(s(x,i,j),z') \leftrightarrow sumlist(s(x,i,j+1),z'')) \leftarrow elem(u,i,j+1,x),\ add(u,i,j,z',z'')$$

Observe that this is now an *S-conditional equivalence*.

Sufficient preliminaries have now been established to allow the following derivation of a recursive procedure for $sumlist^{\dagger}$:-

$\leftarrow sumlist^{\dagger}(x,\hat{i},\hat{j},z,i,j,z')$

$\leftarrow size(x,\hat{i},\hat{j}),\ sumlist^{**}(x,z,s(x,i,j),z')$ *[modus tollens]*

$\leftarrow size(x,\hat{i},\hat{j}),\ (sumlist(x,z) \leftarrow sumlist(s(x,i,j),z'))$

$\leftarrow size(x,\hat{i},\hat{j}),\ (sumlist(x,z) \leftarrow sumlist(s(x,i,j+1),z''),\ elem(u,i,j+1,x),\ add(u,i,j,z',z''))$

$\leftarrow sumlist^{\dagger}(x,\hat{i},\hat{j},z,i,j+1,z''),\ elem(u,i,j+1,x),\ add(u,i,j,z',z'')$

This provides a procedure for $sumlist^{\dagger}$ which deals with the case where $j<\hat{j}$ in the current state of the inspected substructure of *x*. An alternative procedure can be derived for the other case where $j=\hat{j}$, at which point the row coordinate for the next selected element is increased by *1*. Derivation of a basis is trivial, dealing with the case where there remain no more elements to be inspected. The complete procedure set for $sumlist^{\dagger}$ is as follows :-

$sumlist^{\dagger}(x,\hat{i},\hat{j},\hat{z},\hat{i},\hat{j},\hat{z}) \leftarrow$

$sumlist^{\dagger}(x,\hat{i},\hat{j},z,i,\hat{j},z') \leftarrow elem(u,i+1,1,x),\ add(u,i,1,z',z''),\ sumlist^{\dagger}(x,\hat{i},\hat{j},z,i+1,1,z'')$

$sumlist^{\dagger}(x,\hat{i},\hat{j},z,i,j,z') \leftarrow elem(u,i,j+1,x),\ add(u,i,j,z',z''),\ sumlist^{\dagger}(x,\hat{i},\hat{j},z,i,j+1,z'')$

Given suitable means of implementing the calls to *elem* and *add*, these procedures will give excellent iterative behaviour in response to a goal of the form $\leftarrow sumlist^{\dagger}(x,\hat{i},\hat{j},\hat{z},1,0,z^{o})$, proceeding in exactly the same manner as the conventional representation shown earlier. The recursive $sumlist^{\dagger}$ procedures are closely akin to the loop invariants which one might construct in an axiomatization of the latter representation. The computation given for the example portrayed previously using Prolog-like control is as follows, where a term representation is chosen for the output list and the calls to *elem* and *add* have been omitted below just for clarity :-

				element selected
$\leftarrow sumlist^{\dagger}(x,3,5,$	$\hat{z},$	$1,0,$	$0.0.0.0.0.0.0.nil\ )$	
$\leftarrow sumlist^{\dagger}(x,3,5,$	$\hat{z},$	$1,1,$	$3.0.0.0.0.0.0.nil\ )$	3
↓		$1,2,$	$3.1.0.0.0.0.0.nil\ )$	1
↓		$1,3,$	$3.1.4.0.0.0.0.nil\ )$	4
↓		$1,4,$	$3.1.4.4.0.0.0.nil\ )$	4
↓		$1,5,$	$3.1.4.4.2.0.0.nil\ )$	2
↓		$2,1,$	$3.1.4.4.2.0.0.nil\ )$	0
↓		$2,2,$	$3.1.5.4.2.0.0.nil\ )$	1
↓		$2,3,$	$3.1.5.7.2.0.0.nil\ )$	3
↓		↓	↓	↓
↓		↓	↓	↓
$\leftarrow sumlist^{\dagger}(x,3,5,$	$\hat{z},$	$3,4,$	$3.1.12.12.5.10.0.nil\ )$	2
$\leftarrow sumlist^{\dagger}(x,3,5,$	$\hat{z},$	$3,5,$	$3.1.12.12.5.10.9.nil\ )$	9

□ $\hat{z} :=$ *3.1.12.12.5.10.9.nil*

Note that, in addition to an improvement in efficiency, we have also secured much greater clarity in the logic program text.

6.6 : THE EIGHT QUEENS PROBLEM

Specifying the Problem

The eight queens problem is the problem of finding a way of positioning eight chess queens on a conventional chess-board such that none can be taken by any of the others. More generally, we might wish to place *n* pieces of any kind on an *mxm* board subject to any given constraint of interest.

The specific problem of placing eight queens on an *8x8* board has been discussed a great deal in the literature of conventional programming methodology. The usual algorithm employed pursues a potentially exhaustive search through all possible *8*-queen configurations under the control of a backtracking strategy. In deterministic programming languages this arrangement has to be explicitly encoded within the program text, and is consequently a significant challenge to the precept of 'structured programming' which requires control information to be expressed in a clear way using some minimal set of primitive control constructs; this is why the problem appears so frequently in the structured programming literature, its most notable first occurrence in that context being provided by Dijkstra (18). Whilst the task of programming the algorithm in deterministic languages is non-trivial, the problem can be expressed trivially in logic; this just reflects the fact that specification of the problem itself is trivial - it is the algorithm's control information which, in other languages, makes it appear more complicated than it really is. The backtracking strategy is, of course, already inherent in logic program interpreters, so that the programmer has no need to describe it in the course of devising a suitable logic program. Moreover, the logic programmer has no need to re-write specific arrangements for backtracking in the course of writing programs for different problems which might also require that kind of control. He therefore has no need of elaborate tools like the 'control structure abstraction methodology' proposed by Gerhart and Yelowitz *[IEEE Trans. Soft. Eng., SE-2 No.2, 1976]* for expressing the control mechanisms in the eight queens algorithm and in algorithms for other combinatorial problems; the very use of a logic interpreter already comprises such a methodology.

A Horn clause formulation of the eight queens problem is given in an early report by Hogger (36), but without proof that it conforms to an intuitive FOPL specification. Here we derive a procedure set which conforms to a reasonably obvious specification set which simply describes the properties of the desired configuration.

Admit to S the sentence :-

$$config(x,w,z) \leftrightarrow numpos(x,w),$$
$$(\forall u)(onboard(u,z) \leftarrow pos(u,x)),\ neutral(x)$$

in which $config(x,w,z)$ holds when x is a configuration (that is, a set) of w positions on a zxz board such that no piece placed upon any of those positions can take any piece in one of the other positions. The predicate $numpos(x,w)$ expresses the fact that the configuration x contains w positions, $pos(u,x)$ holds when u is a position in the configuration x and $neutral(x)$ means that no pieces in distinct positions in x can take each other. The *neutral*-ity of a configuration can be specified in turn by a further sentence :-

$$neutral(x) \leftrightarrow (\forall u_1 u_2)(notake(u_1,u_2) \leftarrow pos(u_1,x),\ pos(u_2,x),\ u_1 \neq u_2)$$

in which $notake(u_1,u_2)$ holds when a piece on position u_1 cannot take a piece on position u_2. A useful and slightly more abstract way of expressing the *neutral* specification employs the predicate $inviolate(u,x)$ which means that a piece on position u cannot take pieces on any other position in configuration x. Specifications for *neutral* and *inviolate* are given by :-

$$neutral(x) \leftrightarrow (\forall u)(inviolate(u,x) \leftarrow pos(u,x))$$
$$inviolate(u,x) \leftrightarrow (\forall u')(notake(u,u') \leftarrow pos(u',x),\ u \neq u')$$

Synthesis of a suitable procedure set for solving calls to *config* only requires some simple facts about the constructibility of configurations, together with some constraints which specialize the problem to dealing with chess queens on a zxz board. Suppose that any computed configuration is generated by extending a given configuration x' by adding a new position u' to it. This can be expressed by a predicate $extend(u',x',x)$ having the properties :-

$$(\forall u)(pos(u,x) \leftrightarrow pos(u,x') \vee u=u') \leftarrow extend(u',x',x)$$
$$(numpos(x,w) \leftrightarrow numpos(x',w-1)) \leftarrow extend(u',x',x)$$

In addition the empty configuration x satisfies :-

$$(\forall u)(pos(u,x) \leftrightarrow false) \leftarrow numpos(x,0)$$

Enough information is now available to begin a derivation for *config*.

Program for the Eight Queens Problem

To obtain a complete procedure set for *config* it is only necessary to investigate the two alternative ways in which x can be a configuration : either it is empty, or else it is constructible using a call to *extend*. The derivation for the first case is trivial and just gives the procedure :-

$$config(x,0,z) \leftarrow numpos(x,0)$$

The derivation for the second case proceeds as follows :-

$\leftarrow config(x,w,z)$

$\leftarrow numpos(x,w),\ (\forall u)(onboard(u,z) \leftarrow pos(u,x)),\ neutral(x)$

[modus tollens. Note that onboard(u,z) just means that u is a valid position on a zxz board]

$\leftarrow numpos(x',w-1),\ (\forall u)(onboard(u,z) \leftarrow pos(u,x') \vee u=u'),\ neutral(x),\ extend(u',x',x)$

[the usual S-cond.-equiv. substitution]

$\leftarrow numpos(x',w-1),\ (\forall u)(onboard(u,z) \leftarrow pos(u,x')),\ onboard(u',z),\ neutral(x),\ extend(u',x',x)$

[simplification by distribution]

Next we activate the call *neutral(x)* , but for concise presentation this is shown below as a separate derivation :-

$\leftarrow neutral(x)$

$\leftarrow (\forall u)(inviolate(u,x) \leftarrow pos(u,x))$

$\leftarrow (\forall u\hat{u})(notake(u,\hat{u}) \leftarrow pos(u,x),\ pos(\hat{u},x),\ u \neq \hat{u})$

[S-equiv. substitution and simplifying]

$\leftarrow (\forall u\hat{u})(notake(u,\hat{u}) \leftarrow (pos(u,x') \vee u=u'),\ u \neq \hat{u},\ (pos(\hat{u},x') \vee \hat{u}=u')),\ extend(u',x',x)$

[S-cond.-equiv.substitutions]

$\leftarrow (\forall u)(notake(u,u') \leftarrow pos(u,x'),\ u \neq u'),\ neutral(x'),$
$(\forall u)(notake(u',u) \leftarrow pos(u,x'),\ u' \neq u),\ extend(u',x',x)$

[simplifying by distribution and 1:1 property of =]

If this derivation is then assimilated into that for *config*, the definiens of *config(x',w-1,z)* is recognized as a conjunction of the calls *numpos(x',w-1)*, *(∀u)(onboard(u,z) ← pos(u,x'))* and *neutral(x')* and so can be replaced accordingly; thus the following procedure is inferred :-

$$config(x,w,z) \leftarrow config(x',w\text{-}1,z),\ (\forall u)(notake(u,u') \leftarrow pos(u,x'),u\neq u'),\ (\forall u)(notake(u',u) \leftarrow pos(u,x'),u'\neq u),\ onboard(u',z),\ extend(u',x',x)$$

Further simplification at this stage depends upon the provision of knowledge about *onboard* and *notake*, which means that the derivation is to be specialized to deal with a particular class of problems. In the case of the chess queens problem, the meaning of *onboard* is given by :-

$$onboard(p(z_1,z_2),z) \leftrightarrow 1\leq z_1\leq z,\ 1\leq z_2\leq z$$

and the meaning of *notake* by :-

$$notake(p(z_1,z_2),p(z_1',z_2')) \leftrightarrow z_1\neq z_1', z_2\neq z_2',\ |z_1-z_1'|\neq|z_2-z_2'|$$

where the term $p(z_1,z_2)$ represents a position with row coordinate z_1 and column coordinate z_2. Then it is clear that the *notake* relation is symmetric, so that one of the non-atomic calls can be deleted by virtue of being implied by the other. The non-atomic call which then remains is just the definiens of *inviolate(u',x')* and so can be replaced accordingly. If the conventional term representation is used to signify sets of positions (configurations), thus determining that *S* implies the assertions :-

$$extend(u',x',u':x') \leftarrow$$
$$numpos(\emptyset,0) \leftarrow$$

then the above call to *extend* can be eliminated to give the following procedure set for *config* :-

$$config(\emptyset,0,z) \leftarrow$$
$$config(p(z_1,z_2):x',w,z) \leftarrow config(x',w\text{-}1,z),\ 1\leq z_1\leq z,\ 1\leq z_2\leq z,\ inviolate(p(z_1,z_2),x')$$

The recursive procedure can be interpreted informally as follows : given some neutral set of positions x' comprising a partial configuration, this can be extended by adding a new position $p(z_1,z_2)$ provided that this position is on the board (that is, its row and column coordinates are within the bounds *1* and *z*) and that a piece placed upon it cannot take pieces on any of the positions in x'.

A procedure set for *inviolate* is trivially derivable from the given specification; given some position *u* and some configuration *x*, it just iteratively checks whether each piece on a position in *x* is untakeable by *u* :-

```
inviolate(u,x) ← numpos(x,0)
inviolate(u,x) ← extend(u',x',x), notake(u,u'),
                 inviolate(u,x')
```

The necessary derivations just exploit the knowledge in *S* about how configurations are constructed. The calls to *numpos* and *extend* can obviously be easily eliminated by macroprocessing as in the procedures for *config*. A complete set of procedures for solving a goal ← *config(x,8,8)* then consists of the macroprocessed procedures for *config* and *inviolate*, together with the single procedure for *notake* :-

$$notake(p(z_1,z_2),p(z_1',z_2')) \leftarrow z_1 \neq z_1', z_2 \neq z_2', |z_1-z_1'| \neq |z_2-z_2'|$$

Executed with Prolog-like control, they quickly recurse on the *config* procedure until activating a call which establishes the empty configuration. Then partial configurations are generated in an essentially bottom-up fashion as the stacked calls to the two selection procedures for z_1 and z_2 are gradually processed. Each time some new position is returned from these calls, a top-down iterative computation is activated from the call to *inviolate* to test whether that position can be added to the current partial configuration; if not, the computation backtracks to re-invoke the selection procedures in order to find an alternative position.

The arrangement of the selection of candidate positions in the procedures above is inefficient in that no analysis of the current partial configuration is conducted in order to assist the intelligent determination of a new position to add to it. The conventional

remedy for this inefficiency is to arrange the selection of column coordinates such that any k^{th} position in the current partial configuration x' has k as its column coordinate. Then to extend x' by one position to produce x, the new position u' is selected as $p(z_1,w)$ for some $1 \leqslant z_1 \leqslant z$ where w is the number of positions which x will then have. Because the column coordinate of u' in x is then certainly distinct from all column coordinates in x', the inviolability of u' with respect to x' can be investigated by comparisons of row and diagonal coordinates only. The procedures which put these ideas into effect can be obtained by some simple transformations upon those above, or else by backtracking through the derivations to a point at which it is convenient to introduce the assumption that configurations are to be constructed in this restricted way. They are shown below without proof :-

$$config^*(\emptyset,0,z) \leftarrow$$

$$config^*(p(z_1,w):x',\ w,z) \leftarrow config^*(x',w-1,z),\ 1 \leqslant z_1 \leqslant z,\ inviolate^*(p(z_1,w),x')$$

$$inviolate^*(u',\emptyset) \leftarrow$$

$$inviolate^*(u',\hat{u}:\hat{x}) \leftarrow notake^*(u',\hat{u}),\ inviolate^*(u',\hat{x})$$

$$notake^*(p(z_1,z_2),p(z_1',z_2')) \leftarrow z_1 \neq z_1',\ |z_1-z_1'| \neq |z_2-z_2'|$$

It can be assumed that the interpreter can directly solve the calls in the new *notake** procedure, and that the selection call $1 \leqslant z_1 \leqslant z$ is implemented such as to non-deterministically select values of z_1 in the range 1 to z; this is the source of the program's inherent non-determinism, and hence the cause of backtracking in its behaviour with a Prolog-like interpreter. Solution of a goal $\leftarrow config(x,8,8)$ will now be accomplished with efficiency comparable to the conventional representation of the algorithm. The text of the program is clearly very simple by comparison with typical renderings in Algol-like languages. Moreover, the derivation of the procedures required only a trivial analysis of the structure of configurations. The most notable point made by the treatment given here is that the program, its specification and its derivation are all logically innocuous; this just reflects the power of the logic programming formalism in allowing the composition of programs which possess no explicit control information, thereby revealing their simple logic.

CHAPTER 7

TRANSFORMATION OF LOGIC PROGRAMS

PREVIEW

The problems examined in this chapter are rather more difficult than those considered in Chapter 6. Here the aim is to show that the inference rules employed for derivation from specification sets can also be employed to transform programs, for example, to improve efficiency or to achieve a different distribution of logic and control in the components of a particular algorithm.

Two algorithm families are presented. The first consists of some closely related sorting algorithms - the bi-partition sorts. Kowalski's naive-sort is derived first from a very general specification and its behaviour discussed. Naive-sort then forms the basis for an alternative derivation which leads to the general merge-sort. Rather than pursuing other algorithms from scratch in the same way, they are now obtained by specializing merge-sort in various ways. These specializations are all effected by conditional-equivalence substitutions applied to procedures rather than to goals.

The second algorithm family deals with the text searching problem, and is rather more interesting than the sorting algorithms. Here the naive quadratic algorithm is examined in great detail and a number of alternative representations are given for it. The most explicitly informed of these is deterministic, iterative, has explicit provision for matching failures, and gives direct access to the members of both text string and keyword through the use of pointers in its procedures' argument structures. This representation then forms the basis for

some rather subtle transformations on the procedure responsible for responding to a mismatch, leading to the Knuth-Morris-Pratt linear algorithm in one case, and to the Boyer-Moore sub-linear algorithm in another. The reader who is new to logic programming should find in these examples much interesting material illustrating alternative programming styles and the logical relationships between them.

7.1 : LOGIC PROGRAMS FOR SORTING

Sorting and Logic Programming

Logic programs for sorting sets into ordered lists have been studied ever since the inception of logic as a programming language. Early investigations of Horn clause sorting programs were undertaken by van Emden and by Kowalski (47), who examined the algorithm named in this chapter as 'naive-sort'. Kowalski gives a fairly detailed account of naive-sort in his IFIP paper (50), and compares it with a logic program for Hoare's 'quick-sort' in other reports (49,51). Quick-sort is also chosen as an example in van Emden's paper (23).

Although the computational analysis of sorting algorithms has been studied in great depth (reviewed in detail by Knuth's treatise (43) on sorting and searching), formal syntheses for those algorithms have only been pursued comparatively recently. Automated syntheses of sorting programs are reported by Green (31) and by Green and Barstow (32), who use an implementation of a large data base of rules describing fundamental properties of sets, arrays, permutations and ordered lists together with schemas representing simple algorithms for processing those kinds of data structures. Their approach is intended to decide matters of both logic and control, which they do not separately consider and represent as we do in our treatment of logic program synthesis.

Darlington (20) has also presented syntheses of several sorting algorithms expressed in a recursion equation language, based upon the ideas underlying his transformation system reported variously in (9), (10) and (19). This system is semi-automatable to the extent that whilst an interacting user decides upon suitable definitions for the functions of interest, together with prescriptions for their subsequent manipulation, the mechanized part of the implementation assumes responsibility both for preserving correctness and for contributing in a limited but useful way towards the exploration of the search space determined by the input definitions and the transformation rules. However, Darlington considered that semi-automation of his sorting program syntheses as originally formulated was impractical; his paper presents them as examples of wholly non-mechanical syntheses.

Derivation of sorting programs presented a useful challenge to the development of methods for logic program synthesis, and so several of Darlington's examples were reformulated in predicate logic by Hogger (38), but employing a much simpler ontology. Darlington's original treatment of sorting introduced quite a lot of rather high-level properties of the functions of interest into the initial problem description, but these do not seem to be necessary in order to derive the essential structure of the required programs. They also tend to obscure some of the more general taxonomic relationships between the various kinds of sorting algorithms, which in fact are capable of clarification in much simpler terms than presented in his report. Subsequently, some of these sorting algorithms were derived by Clark and Darlington (13) using a notation which is a hybrid of recursive function language and Horn clause logic. They emphasize the usefulness of a synthetic approach to the study of algorithm families, observing that much can be learnt about their similarities and differences from identifying critical decision points in a tree of derivations which spans the space of all sorting programs determined by the given axiomatization.

An earlier analysis of a quick-sort logic program was shown in the paper by Clark and Tarnlund (16), who summon it as an example of their treatment of verification described here in Chapter 4. There they begin with a sorting program and then prove various properties about it which a correct sorting program ought to possess. Their treatment there of a proof of quick-sort is also interesting in its use of a somewhat novel data structure representation for the sorted lists produced as the program's output.

The Naive-Sort Algorithm

Naive-sort is the algorithm obtained by applying a naive procedural interpretation to the basic definition of sortedness. Because that definition serves as a starting point for all the sorting programs considered here, it will be useful to discuss it immediately in some detail.

Throughout the present examination of sorting, the predicate of primary interest names a two place relation *sort(x,y)* which

holds between a set x and a list y when y is both a permutation of x and ordered by a total ordering relation $<$. A simple way of specifying orderedness has already been shown in previous examples which uses the *item* predicate as a primitive constructor of lists :-

$$ord(y) \leftrightarrow (\forall uv)(\ u<v \leftarrow consec(u,v,y)\ )$$
$$consec(u,v,y) \leftrightarrow (\exists i)(item(u,i,y),\ item(v,i+1,y))$$

[together with the list axioms A1-A3
and general laws about $<$]

The assumption that $<$ is a total ordering relation (rather than just any binary relation) has the consequence that these axioms can be shown to imply an alternative way of specifying the *ord* relation as follows :-

$$ord(y) \leftrightarrow (\forall uv)(\ u<v \leftarrow prec(u,v,y)\ )$$

when augmented by the definition of *prec* :-

$$prec(u,v,y) \leftrightarrow (\exists ij)(item(u,i,y),\ item(v,j,y),\ i<j)$$

The predicate $prec(u,v,y)$ means that the member u precedes the member v in the list y. The alternative *ord* specification is a consequence of the transitivity of the total ordering $<$. We shall use either specification according to convenience.

The notion of permutedness is expressed here using a predicate $perm(x,y)$ which holds when the list y is a permutation of the members of the set x, and is specified as follows :-

$$perm(x,y) \leftrightarrow (\forall u)(\ u\varepsilon x \leftrightarrow u\hat{\varepsilon}y\ ),$$
$$(\forall u)(\ u\varepsilon x \leftrightarrow occurs(u,1,y)\ )$$

where ε and $\hat{\varepsilon}$ are respectively the set and list membership relations, and $occurs(u,z,y)$ holds when a member u has z distinct occurrences in the list y ; that is, when u has multiplicity z in y. The *sort* relation can then be specified simply by :-

$$sort(x,y) \leftrightarrow perm(x,y),\ ord(y)$$

The interpretation of permutedness adopted above is in accordance with the conventions used by Knuth (43) and by Darlington (20) for defining the *perm* relation, but is at slight variance with the interpretations found in the papers by Kowalski (50), by Clark and Tarnlund (16) and by Clark and Darlington (13). All of these interpret $perm(x,y)$ to mean that a list y is a rearrangement of a

list x which preserves each member's multiplicity. This difference is of some consequence since it determines differing specifications and derivations. Knuth's treatment of multiple occurrences employs a kind of set-list hybrid called a 'multiset'; he generalizes the notion of permutation of a set to permutation of a multiset, and hence arrives at a position to sort multisets into ordered lists. However, if one really does wish to rearrange a list, the appropriate specification is :-

$$perm(x,y) \leftrightarrow (\forall uz)(occurs(u,z,x) \leftrightarrow occurs(u,z,y))$$

which just requires that a member occurring z times in x must occur z times in y. This must, of course, be augmented by a specification for *occurs* and the axioms for well-formed lists. The analyses given by Clark and Tarnlund (16) and by Clark and Darlington (13) have the object of dealing with programs which rearrange lists in this way, but they use the following sentence as their specification :-

$$perm(x,y) \leftrightarrow (\forall u)(\ u\hat{\varepsilon}x \leftrightarrow u\hat{\varepsilon}y\ )$$

It would seem that this sentence does not precisely capture the relation which they actually expect to hold between x and y, because it admits possibilities such as $x = (1,2)$ and $y = (2,2,2,1,1,2)$ in consequence of dropping the constraint upon preservation of multiplicities. The relation which they wish to compute is therefore properly included in the relation which they specify, and so they have to resort to incomplete procedure sets which are only capable of executing the desired rearrangements. This does not appear to be a very satisfactory way of proceeding, even though the use of a weaker specification makes the higher-level derivations easier than they would otherwise be. The treatment here, by contrast, will begin with an accurate specification of the intended computed relation and then pursue complete procedure sets for it; this eliminates the uncertainties which would otherwise prevail about the relationship between the specification and the high-level procedures derived from it.

The naive-sort algorithm, as observed previously, just interprets the sortedness specification in a naive way, searching for complete permutations of the set x until discovering one which is ordered. Its logic component therefore uses the procedures :-

sort(x,y) ← perm(x,y), ord(y)

[together with procedure sets for perm and ord]

whilst its control component is just top-down LIFO (Prolog-like) call scheduling. As a result the calls dealing respectively with permutedness and orderedness are, in the simplest control arrangement of this kind, executed sequentially and independently. This algorithm is clearly very inefficient, generating a comparison count of the order *(n-1).n!* where *n* is the cardinality of *x* (the comparisons arising in the orderedness checks instigated by the call to *ord*). This is much worse than the count $n.log_e n$ which is normally expected of a 'good' sorting algorithm. · The inefficiency arises from the circumstance that when some instance of *y* computed by the call to *perm* fails to pass the check on orderedness, the ensuing backtracking causes the entire permutation to be discarded, even though it might contain large sublists which appear also in the correct solution of *y*. Thus the backtracking destroys knowledge about comparisons which must be generated afresh in checking subsequent instances of *y*. Clark and Kowalski have examined ways of adjusting the control of backtracking in this sorting algorithm and have shown that some improvement in its behaviour can be achieved. Kowalski (50) has also investigated the behaviour obtained by executing the procedures above with a coroutining control strategy, and has shown that this also improves upon naive-sort; it does this by arranging that the choice of new members made in the execution of *perm* whilst constructing partial permutations is constrained by intermittent activation of the *ord* call in order to decide whether that choice will preserve orderedness when the member is appended to give an extended partial permutation. Despite such improvements in the control, the behaviour of the naive-sort logic component cannot, apparently, aspire to that of the commonly used sorting algorithms.

The derivation of procedures for *perm* and *ord* is accomplished by exploiting knowledge about the constructibility of sets and lists. In the case of naive-sort the usual procedures employed for *perm* and *ord* make calls to a partitiioning procedure *partition*(u',x',x)* which just selects an arbitrary member *u'* of *x* to leave *x'*, where *u'* is the next member to be added to the current partial permutation. The latter process is implemented by a call to the familiar procedure *append*(u',y',y)*. Neither of these two calls makes any special assumptions about the relationship between *u'* and the data structures *x'* and *y'*; the *partition** call does not choose *u'* by comparing it in

any way with the other members in x, and the *append** call does not append y' to u' on the assumption that this will preserve orderedness.

The sentences admitted to S in order to derive these procedures are as follows : -

$$append^*(u',y',y) \leftrightarrow (\forall ui)(item(u,i,y) \leftrightarrow item(u,i-1,y') \vee (u=u',i=1))$$

$$partition^*(u',x',x) \leftrightarrow (\forall u)(\ u\varepsilon x \leftrightarrow u\varepsilon x' \vee u=u'\ ),\ (\exists u)u\varepsilon x',\ \sim u'\varepsilon x'$$

$$u\hat{\varepsilon}y \leftrightarrow (\exists i)item(u,i,y)$$

together with list axioms A1-A3 and set axioms asserting that sets either have cardinality 0 or are singletons or are constructible by partition.*

These form a sufficient adjunct to the elementary specifications for *sort*, *perm* and *ord* to allow reasonably straightforward derivations. The latter are not presented here in the case of the *ord* relation because they have already been indicated elsewhere in the thesis and raise no special issues. They are :-

$$ord(y) \leftarrow length(y,0)$$
$$ord(y) \leftarrow length(y,1)$$
$$ord(y) \leftarrow append^*(u',y',y),\ append^*(v',y'',y'),\ u'<v',\ ord(y')$$

The *perm* synthesis, however, requires proofs of some preliminary lemmas in order to obtain the usual recursive *perm* procedure. To show the motivation for summoning those lemmas, consider how the *perm* derivation begins :-

$$\leftarrow perm(x,y)$$
$$\leftarrow (\forall u)(\ u\varepsilon x \leftrightarrow u\hat{\varepsilon}y\ ),\ (\forall u)(\ u\varepsilon x \leftrightarrow occurs(u,1,y)\ )$$

It is required to substitute references to u', x' and y' for references in the goal to x and y in order to explore the consequences of the assumptions about the constructibility of x and y. Note that this will require substitutions for identical occurrences of the predicate $u\varepsilon x$: we must resist the temptation to simplify the goal above to $(\forall u)(\ u\varepsilon x \leftrightarrow u\hat{\varepsilon}y,\ occurs(u,1,y)\ )$ since this is not sound [as a counter-example, consider $x=\emptyset$, $y=(a,a)$ which solves the conjectured simplification but not the goal above]. The necessary lemmas are :-

$(\forall u)(\ u \varepsilon x \leftrightarrow u \varepsilon x' \vee u=u'\ ) \leftarrow partition^*(u',x',x)$

$(\forall u)(\ u \hat{\varepsilon} y \leftrightarrow u \hat{\varepsilon} y' \vee u=u'\ ) \leftarrow append^*(u',y',y)$

$(\forall u)(\ occurs(u,l,y) \leftrightarrow occurs(u,l,y') \vee u=u') \leftarrow append^*(u',y',y),\ \sim u' \hat{\varepsilon} y'$

The first of these follows trivially from the *partition** specification; the second is obtained easily by making an *S-conditional-equivalence substitution* for $item(u,i,y)$ in the definiens of $\hat{\varepsilon}$ conditional upon $append^*(u',y',y)$ and then simplifying the result; and the third is obtained with a little more difficulty by making a similar substitution in the definiens of $occurs(u,l,y)$ using the specification :-

$$occurs(u,l,y) \leftrightarrow (\exists i)(\forall \hat{i})(\ i=\hat{i} \leftrightarrow item(u,\hat{i},y))$$

Applying all three lemmas to the above derivation goal for *perm* in the context of *S-conditional-equivalence substitutions* produces :-

$\leftarrow (\forall u)(\ u \varepsilon x' \vee u=u' \leftrightarrow u \hat{\varepsilon} y' \vee u=u'),$
$(\forall u)(\ u \varepsilon x' \vee u=u' \leftrightarrow occurs(u,l,y') \vee u=u'),\ append^*(u',y',y),$
$partition^*(u',x',x),$
$\sim u' \hat{\varepsilon} y'$

$\leftarrow perm(x',y'),\ append^*(u',y',y),\ partition^*(u',x',x),\ \sim u' \hat{\varepsilon} y'$

[simplification by cancellation of identical disjuncts, and then definiens replacement]

$\leftarrow perm(x',y'),\ append^*(u',y',y),\ partition^*(u',x',x)$

[because it is easy to show that the last call is implied by the conjoined calls to perm and partition]*

Derivations for *perm* bases are trivial, and just exhaust the remaining cases of the structures of x and y. The complete procedure set for *perm* used by naive-sort is then as follows :-

$perm(x,y) \leftarrow cardin(x,0),\ length(y,0)$
$perm(x,y) \leftarrow singleton(x,u),\ item(u,1,y),\ length(y,1)$
$perm(x,y) \leftarrow partition^*(u',x',x),\ perm(x',y'),$
$append^*(u',y',y)$

Macroprocessing the procedure sets for *perm* and *ord* by choosing the usual term representations will then render them in a form rather

like those which appear in Kowalski's examples (49,50). A suitable procedure set for *partition** is trivially derivable. The complete naive-sort program body for processing sets and lists represented by terms is as follows :-

sort(x,y) ← perm(x,y), ord(y)

perm(∅,nil) ←
perm(u:∅,u.nil) ←
perm(x,u'.y') ← partition(u',x',x), perm(x',y')*

ord(nil) ←
ord(u.nil) ←
ord(u.v.y") ← u<v, ord(v.y")

partition(u',x',u':x') ← u'∉x'*
partition(u',v:x",v:x') ← partition*(u',x",x')*

u'∉∅ ←
u'∉v:x' ← u'≠v, u'∉x'

Observe that the call $u' \notin x'$ in the first *partition** procedure acts essentially as a type-check upon the input term representing the set to be partitioned. If the call is deleted - so that the resulting *partition** procedures no longer describe the true set partitioning relation - then the program can sort multisets into ordered lists and thus behave essentially the same as Kowalski's program for finding ordered rearrangements of arbitrary input lists.

This discussion of naive-sort closes by showing an interesting transformation which employs just the procedures above and S ; no other information is necessary in order to put it into effect. Suppose, generally, that we have two procedures $A_1 \leftarrow B_1$ and $A_2 \leftarrow B_2$; then these jointly imply a new procedure $(A_1,A_2) \leftarrow (B_1,B_2)$. If this fact is exploited with the procedures above, coupling each *perm* procedure with its counterpart for *ord*, the result is :-

sort(∅,nil) ←
sort(u:∅,u.nil) ←
sort(x,u.v.y") ← partition(u,x',x), sort(x',v.y"), u<v*

[together with procedures solving the call to partition]*

With Prolog-like control these behave quite like the naive-sort procedures executed using coroutining as mentioned a little earlier.

The Merge-Sort Algorithm

In the naive-sort synthesis the relations *partition** and *append** were specified in order to express just one way of constructing the sets and lists in question. Neither of them incorporated any assumptions about the relative magnitudes of the constituent members of the data structures which they related. In the algorithms now to be introduced the construction of sets and lists is constrained in a variety of ways which reflect just such assumptions. For example, we shall see that merge-sort constructs the output list from two given lists in a manner which takes account of the ordering of the members which are manipulated in the course of that construction. By contrast, quick-sort partitions the input set in a manner which decides the particular partitioning by comparing the input set's members. From a very general point of view, we may say that such arrangements allow the decomposition of the input set and the composition of the output list to be more informed about their contributions towards the ultimate goal of generating a complete ordered permutation than was the case in the strategy employed by naive-sort.

Merge-sort belongs to a family of algorithms which may be loosely described as 'bi-partition sorts'. Their characteristic feature is that the input set x is sorted to give the output list y by bi-partitioning x into two subsets x_1 and x_2, sorting these to give ordered lists y_1 and y_2, and finally constructing y from y_1 and y_2. During this process various pairs of members originating from x are compared to provide knowledge about their eventual relative positions in y.

Algorithms in this family can be arranged within a spectrum, at one extreme of which are those which defer all comparisons until both x_1 and x_2 have been computed, whilst those at the other extreme perform no comparisons after x_1 and x_2 have been computed. Merge-sort is of the former kind : x is partitioned arbitrarily into x_1 and x_2, so that construction of y cannot proceed without comparing members of y_1 with y_2. At the opposite extreme is quick-sort : x is partitioned such that every member of x_2 exceeds every member of x_1, after which y is constructed from y_1 and y_2 with no further comparisons. Both of these algorithms are conceptually simple, although their practical implementations - like those of any other sorting algorithm -

may only secure these manipulations of the data structures in deeply implicit ways, for example by allowing them to share common memory and controlling the movement of members by elaborate pointer systems. Certain rather more complicated algorithms , like Williams' heap-sort (88), occupy intermediate positions in this notional spectrum. Heap-sort bi-partitions x into x_1 and $x_2 = \{u\} \cup x_2'$ using enough comparisons to establish a representation of x as an ordered tree with root node u and sub-trees representing x_1 and x_2' ; the algorithm then initiates an elaborate 'sifting' computation which constructs y by making further comparisons between members of the tree, implicitly constructing and simultaneously merging the ordered permutations of x_1 and x_2 as it does so. Floyd's variant of heap-sort (25) is partially derived in the paper by Burstall and Darlington (10).

In contemplating a logic synthesis for merge-sort, it is useful to briefly reconsider naive-sort, whose fundamental failing is that no comparisons are made during the construction of any candidate permutation y; a permutation is generated first, and then inspected for orderedness. To obtain more sensible behaviour, knowledge about the progress made towards the solution at any stage during the computation must be made accessible to whatever permuting and ordering activities still remain to be accomplished, so that they may proceed more intelligently than they otherwise would. An improvement of this kind can be obtained through the agency of the logic component by arranging particular ways of permuting and ordering which are conditional upon other constraints controlling knowledge about the relative magnitudes of members. The conditional-equivalence substitution is ideal for this purpose during the derivation of procedures intended to behave in this way; for instance, it will be seen that the calls to *perm* and *ord* in the preliminary derivation goal can be usefully elaborated by subformula substitution to reveal special ways of solving them which are conditional upon the new call introduced by that inference rule; that call will inform the goal about the permutedness and orderedness of the subsets and sublists from which the input and output lists are composed, and will thereby dispose the derivation to a particular kind of algorithm which improves upon naive-sort. In fact the merge-sort synthesis can be seen as a symbolic execution of naive-sort, beginning with its general goal $\leftarrow$ *perm(x,y), ord(y)* but solving the calls using different facts about sorting than those which normally comprise the naive-sort program.

The specification set chosen for the merge-sort derivation defines the *sort* relation exactly as for naive-sort :-

$$sort(x,y) \leftrightarrow perm(x,y),\ ord(y)$$

The *perm* specification is also exactly as before, together with associated axioms characterizing sets and lists. Now the merge-sort algorithm, as already explained, assumes that the input set can be partitioned arbitrarily, leaving the task of constructing output until the latter has been completed. In view of this, it is appropriate to introduce to S the relation which expresses this particular way of decomposing x :-

$$\begin{aligned} partition(x_1,x_2,x) \leftrightarrow\ & (\forall u)(\ u\varepsilon x \leftrightarrow u\varepsilon x_1 \vee u\varepsilon x_2\ ),\\ & (\forall u)(\ u\varepsilon x_1,\ u\varepsilon x_2 \leftrightarrow false),\\ & (\exists u)(\ u\varepsilon x_1),\ (\exists u)(\ u\varepsilon x_2) \end{aligned}$$

where $partition(x_1,x_2,x)$ holds when x is partitioned into the two disjoint, non-empty subsets x_1 and x_2.

When x_1 and x_2 are chosen arbitrarily and then sorted to give y_1 and y_2 respectively, the output y has to be computed by interleaving y_1 with y_2 such as to achieve orderedness. This process is said to 'merge' y_1 and y_2. The specification of merging needs a little care. Here we adopt Knuth's interpretation (43) which considers that *"Merging means the combination of two or more <u>ordered</u> files into a single ordered file."* Therefore the *merge* specification given here explicitly requires that y_1 and y_2 shall both be ordered. This position is somewhat different from that taken by Clark and Darlington (13), who choose instead a slightly strange meaning for the concept of 'merging' y_1 and y_2 to give y : they require that y shall be a permutation of the members composing y_1 and y_2, and that y shall be ordered if both y_1 and y_2 are ordered. This means that they do not consider that the result of merging two lists must be unique, for in the event that one or both are unordered, they allow the result of the merging to be any permutation of their members; this they incorrectly describe as a merging 'function' from (y_1,y_2) to y. They do not take their analysis as far as deriving the lower-level procedures for their programs, such as for *merge*, and so the consequences of adopting their treatment are not clear. However, it would seem that by choosing their particular specification of *merge*, they would eventually find themselves in the position of either having to construct an incomplete procedure set for *merge* (omitting the procedures necessary for merging unordered

lists, and so having to incorporate assumptions of orderedness into analyses such as proof of termination) or else having to construct a complete but non-deterministic procedure set in any completeness-preserving derivation methodology which they might employ. They arrive in this position, not by deliberately choosing such a specification, but rather through manipulating a specification of sortedness until recovering a subformula which they then interpret as a definiens for *merge* ; from our point of view here, that subformula is a *consequence* of, but not an *instance* of, the customary definiens of the merging function.

In view of the considerations above, then, the specification chosen here for *merge* is as follows :-

$$\begin{aligned} merge(y_1,y_2,y) \leftrightarrow\ & ord(y_1),\ ord(y_2), \\ & (\forall uv)(prec(u,v,y) \leftrightarrow prec(u,v,y_1) \vee prec(u,v,y_2) \\ & \qquad \vee u{\leq}v,(spans(u,v,y_1,y_2) \vee \\ & \qquad\qquad spans(v,u,y_1,y_2))), \\ & (\forall uw)(occurs(u,w,y) \leftrightarrow (\exists w_1 w_2)(occurs(u,w_1,y_1), \\ & \qquad\qquad occurs(u,w_2,y_2), \\ & \qquad\qquad w_1{+}w_2 = w)) \end{aligned}$$

Informally, this says that the merge relation holds when both y_1 and y_2 are ordered, composition of y from them preserves ordering, and each member in y has as many occurrences in y as it has jointly in y_1 and y_2. The predicate $spans(u,v,y_1,y_2)$ just summarizes $(u\hat{\varepsilon}y_1,\ v\hat{\varepsilon}y_2)$.

Finally a specification of orderedness must be given. Here we choose the alternative specification :-

$$ord(y) \leftrightarrow (\forall uv)(\ u{<}v \leftarrow prec(u,v,y))$$

[and the spec. for prec]

because it will be required during the derivation to investigate the orderedness of y conditional upon the assumption that it is constructed by $merge(y_1,y_2,y)$, and it will clearly be convenient to arrange, as above, that both ideas refer to the relative positions of members (in terms of precedence rather than consecutivity) in the same way; *merge* is very hard to specify using *consec* to describe the relative positions of members in the output list.

The specification set now contains enough information to begin a synthesis for the *sort* relation. This is most conveniently represented as the combination of derivations from the calls *perm(x,y)* and *ord(y)* in the derivation goal :-

$\leftarrow$ *sort(x,y)*
$\leftarrow$ *perm(x,y), ord(y)*

just as in the previous presentation of the naive-sort synthesis. Once again, the definiens' of *perm* and *ord* are introduced to the goal by *modus tollens* in order to reveal their dependence upon relations over the members of *x* and *y*. We pursue a *perm* derivation first, this being the harder one. For this, some lemmas will be needed as in naive-sort in order to exploit the assumptions that *x* and *y* are respectively constructed by *partition* and *merge*. These lemmas are :-

$$(\forall u)(\ u\varepsilon x \leftrightarrow u\varepsilon x_1 \vee u\varepsilon x_2\ ) \leftarrow partition(x_1,x_2,x)$$
$$(\forall u)(\ u\hat{\varepsilon}y \leftrightarrow u\hat{\varepsilon}y_1 \vee u\hat{\varepsilon}y_2\ ) \leftarrow merge(y_1,y_2,y)$$
$$(\forall u)(\ occurs(u,1,y) \leftrightarrow (occurs(u,1,y_1),\ occurs(u,0,y_2)) \vee (occurs(u,0,y_1),\ occurs(u,1,y_2))) \leftarrow merge(y_1,y_2,y)$$

The first lemma is just a trivial consequence of the *partition* specification. The second one is obtained by observing the fact that $merge(y_1,y_2,y)$ implies its fourth definiens conjunct and that this in turn implies $(\forall u)(\ u\hat{\varepsilon}y \leftrightarrow u\hat{\varepsilon}y_1 \vee u\hat{\varepsilon}y_2\ )$ by virtue of the simple relationship assumed in the list axioms of *S* :-

$$u\hat{\varepsilon}y \leftrightarrow (\exists w)(occurs(u,w,y),\ w>0)$$

together with elementary properties of =, > and + . The third lemma is just a consequence of $merge(y_1,y_2,y)$ implying the instance of its fourth definiens conjunct in the case *w:=1*, together with properties of = and + . Equipped with these, the derivation from $\leftarrow$ *perm(x,y)* is :-

$\leftarrow (\forall u)(\ u\varepsilon x \leftrightarrow u\hat{\varepsilon}y\ )\ .\ (\forall u)(\ u\varepsilon x \leftrightarrow occurs(u,1,y))$
$\leftarrow (\forall u)(\ u\varepsilon x_1 \vee u\varepsilon x_2 \leftrightarrow u\hat{\varepsilon}y_1 \vee u\hat{\varepsilon}y_2),$
$(\forall u)(\ u\varepsilon x_1 \vee u\varepsilon x_2 \leftrightarrow (occurs(u,1,y_1),\ occurs(u,0,y_2)) \vee (occurs(u,0,y_1),\ occurs(u,1,y_2))),$
$partition(x_1,x_2,x),\ merge(y_1,y_2,y)$

$\leftarrow perm(x_1,y_1),\ perm(x_2,y_2),$
$(\forall u)(\ u\varepsilon x_1 \leftrightarrow occurs(u,1,y_1),\ occurs(u,0,y_2)),$
$(\forall u)(\ u\varepsilon x_2 \leftrightarrow occurs(u,0,y_1),\ occurs(u,1,y_2)),$
$partition(x_1,x_2,x),\ merge(y_1,y_2,y)$

[using some simple distributions to simplify, followed by obvious definiens replacements]

The goal at this stage clearly reflects the general idea of the merge-sort algorithm; it describes how permutedness is achieved when x and y are constructed by *partition* and *merge*. The two non-atomic calls can be deleted by virtue of being implied by the others. To show this, note that S implies :-

$$occurs(u,0,y_i) \leftrightarrow \sim u\hat{\varepsilon}y_i$$

and that the specifications of *perm* and *partition* trivially imply :-

$(\forall u)(\ \sim u\hat{\varepsilon}y_2 \leftarrow occurs(u,1,y_1)\ ),$
$(\forall u)(\ \sim u\hat{\varepsilon}y_1 \leftarrow occurs(u,1,y_2)\ ) \leftarrow perm(x_1,y_1),\ perm(x_2,y_2),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad partition(x_1,x_2,x)$

These facts determine immediately that the subformulas $occurs(u,0,y_i)$ can be deleted from the goal above, and then the reduced non-atomic calls are implied by the respective two calls to *perm*. Thus the goal simplifies to one with wholly atomic calls.

To obtain a complete procedure set for *perm* it is only necessary to consider the cases where x and y are not constructible by *partition* and *merge* . These are just the cases where x is empty or a singleton, and the derivations are too trivial to present here. The final procedure set is :-

$perm(x,y) \leftarrow cardin(x,0),\ length(y,0)$
$perm(x,y) \leftarrow singleton(x,u),\ item(u,1,y),\ length(y,1)$
$perm(x,y) \leftarrow partition(x_1,x_2,x),\ perm(x_1,y_1),\ perm(x_2,y_2),$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad merge(y_1,y_2,y)$

The other half of the *sort* synthesis deals with the call to *ord* in the original derivation goal. Again, the bases are trivial and so will be given without proof. The derivation of the recursive procedure for *ord* is somewhat more interesting and proceeds as follows :-

$\leftarrow ord(y)$

$\leftarrow (\forall uv)(\ u<v \leftarrow prec(u,v,y))$

$\leftarrow (\forall uv)(\ u<v \leftarrow prec(u,v,y_1) \vee prec(u,v,y_2) \vee u \leqslant v, (spans(u,v,y_1,y_2) \vee spans(v,u,y_1,y_2))),$

$merge(y_1,y_2,y)$

$\leftarrow (\forall uv)(\ u<v \leftarrow prec(u,v,y_1)), (\forall uv)(\ u<v \leftarrow prec(u,v,y_2)),$
$(\forall uv)(\ u<v \leftarrow u \leqslant v, (spans(u,v,y_1,y_2) \vee spans(v,u,y_1,y_2))),$
$merge(y_1,y_2,y)$

$\leftarrow ord(y_1), ord(y_2), (\forall u)(\ u<u \leftarrow spans(u,u,y_1,y_2)), merge(y_1,y_2,y)$

$\leftarrow ord(y_1), ord(y_2), merge(y_1,y_2,y), partition(x_1,x_2,x),$
$perm(x_1,y_1), perm(x_2,y_2)$

[because S trivially implies :-

$\sim spans(u,u,y_1,y_2) \leftarrow partition(x_1,x_2,x),$
$perm(x_1,y_1), perm(x_2,y_2)$ *]*

thus giving a goal with atomic calls only. The three procedures then inferred in this synthesis for *ord* are :-

$ord(y) \leftarrow length(y,0)$
$ord(y) \leftarrow length(y,1)$
$ord(y) \leftarrow partition(x_1,x_2,x), sort(x_1,y_1), sort(x_2,y_2),$
$merge(y_1,y_2,y)$

[Note that the calls to *sort* in the recursive procedure just arise by replacing the conjoined calls to *perm* and *ord* in the final goal.*]*

The procedure sets for *perm* and *ord* can now be combined as in the naive-sort synthesis to give procedures for *sort* :-

$sort(x,y) \leftarrow cardin(x,0), length(y,0)$
$sort(x,y) \leftarrow singleton(x,u), item(u,1,y), length(y,1)$
$sort(x,y) \leftarrow partition(x_1,x_2,x), sort(x_1,y_1), sort(x_2,y_2),$
$merge(y_1,y_2,y)$

These are the high level procedures for the intended merge-sort algorithm : to sort a set with more than one member, partition it into x_1 and x_2, sort these to y_1 and y_2, and finally merge these to give the desired output y.

A complete program body for merge-sort will also require procedure sets for solving the calls to *partition* and *merge*. Procedures which arbitrarily partition sets are very easy to derive, and so will not be considered further here; they are just simple generalizations of the *partition** procedures used in naive-sort. The procedures for *merge* are a little more interesting, and so we review these briefly. The derived *merge* procedures obtained by just assuming minimal information about constructibility with *append** are as follows :-

$$merge(y_1,y_2,y) \leftarrow ord(y_1),\ length(y_2,0)$$
$$merge(y_1,y_2,y) \leftarrow ord(y_2),\ length(y_1,0)$$
$$merge(y_1,y_2,y) \leftarrow ord(y_1),\ append^*(u,y_1',y_1),\ append^*(v,y_2',y_2),\ u \leqslant v,\ merge(y_1',y_2,y'),\ append^*(u,y',y)$$
$$merge(y_1,y_2,y) \leftarrow ord(y_2),\ append^*(u,y_1',y_1),\ append^*(v,y_2',y_2),\ v \leqslant u,\ merge(y_1,y_2',y'),\ append^*(v,y',y)$$

The residual calls to *ord* in these procedures are just consequences of the *merge* specification insisting upon the lists y_1 and y_2 being ordered; this is required irrespective of the context in which the procedures are called. However, in the present context we know that y_1 and y_2 are necessarily ordered before being processed by the call to *merge* in the recursive *sort* procedure, because they have both been transmitted as output from the calls $sort(x_1,y_1)$, $sort(x_2,y_2)$. To have their orderedness subsequently checked by the *merge* procedures would clearly be computationally intolerable. One could simply delete the calls to *ord* from the *merge* procedures, but this would not leave true theorems about *merge* as specified by S even though they would compute the correct output in their present context. However, there is an interesting transformation which is logically justifiable and is tantamount to deleting the unwanted checks on the orderedness of y_1 and y_2. The technique used is similar to that for deriving the linear natural ordering factorial algorithm derived in the last chapter, in that we construct a new predicate which is inherently conditional upon the predicate which we do not want to be explicitly tested at run-time. In the present case, introduce the specification :-

$$merge^*(y_1,y_2,y) \leftrightarrow (merge(y_1,y_2,y) \leftarrow ord(y_1), ord(y_2))$$

Then it is very easy to show that a call to *merge* in which y_1 and y_2 are ordered can be investigated by a call to *merge**. This is because the sentence above trivially implies :-

$$merge(y_1,y_2,y) \leftarrow ord(y_1), ord(y_2), merge^*(y_1,y_2,y)$$

Now observe that the first *merge* basis above can be written as :-

$$merge(y_1,y_2,y) \leftarrow ord(y_1), ord(y_2), length(y_2,0)$$

since the basis for *ord* determines that $ord(y_2)$ is trivially implied by $length(y_2,0)$. Thus the *merge* basis can be rewritten yet again to give :-

$$(merge(y_1,y_2,y) \leftarrow ord(y_1), ord(y_2)) \leftarrow length(y_2,0)$$

But this immediately implies a procedure for *merge** :-

$$merge^*(y_1,y_2,y) \leftarrow length(y_2,0)$$

This is one of two similar bases for a *merge** procedure set. The recursive *merge* procedures can be transformed in a similar fashion. Firstly assume that the following property of *ord* can be easily established :-

$$ord(y_1') \leftarrow append^*(u,y_1',y_1), ord(y_1)$$

Then substitute for $merge(y_1',y_2,y')$ in the first recursive procedure for *merge* above using *modus tollens* in conjunction with the following sentence trivially implied by the *merge** specification :-

$$merge(y_1',y_2,y') \leftarrow ord(y_1'), ord(y_2), merge^*(y_1',y_2,y')$$

The result of this is :-

$$merge(y_1,y_2,y) \leftarrow ord(y_1), append^*(u,y_1',y_1), append^*(v,y_2',y_2), u \leqslant v, ord(y_1'), ord(y_2), merge^*(y_1',y_2,y'), append^*(u,y',y)$$

Now substitute for the predicate $ord(y_1')$ using the property of *ord* assumed above and rearrange the connectives to give :-

$$(merge(y_1,y_2,y) \leftarrow ord(y_1), ord(y_2)) \leftarrow append^*(u,y_1',y_1), append^*(v,y_2',y_2), u \leqslant v, merge^*(y_1',y_2,y'), append^*(u,y',y)$$

and finally replace the consequent formula by an instance of *merge** to give one of two recursive procedures for *merge**. The complete *merge** procedure set is then exactly as though the checks upon

the orderedness of y_1 and y_2 were deleted from the *merge* procedures, and then *merge* renamed as *merge**. The call to *merge* in the recursive *sort* procedure can be replaced simply by $merge^*(y_1,y_2,y)$ because the calls to *sort* in that procedure imply that both y_1 and y_2 are ordered. With top-down Prolog-like control, the merge-sort program body as outlined here will solve a call to *sort* with a reasonable computation (that is, reasonable for a recursive algorithm).

The merge-sort procedures will next be used as the foundation for a quick-sort program. In fact we shall see that a simple *S*-conditional-equivalence substitution is sufficient to transform the principal merge-sort *sort* procedure into one which captures the logic of quick-sort. After that, an alternative transformation is given to turn merge-sort into insert-sort, and finally quick-sort is transformed into selection-sort.

The Quick-Sort Algorithm

Hoare's quick-sort algorithm (35) can be regarded as a specialized case of merge-sort in which the input set is partitioned into sets x_1 and x_2 satisfying the property that every member of x_2 exceeds every member of x_1. In other words, the partitioning process is required to take some responsibility for comparing the members of x. When x is partitioned as above, the relationship between x_1 and x_2 will be summarized by the predicate $smaller(x_1,x_2)$.

Rather than deriving quick-sort from scratch, it is more interesting to explore the consequences of introducing to the existing merge-sort procedures the assumption that $smaller(x_1,x_2)$ holds. Clearly the *sort* bases are not affected by this assumption, and so it is only necessary to consider its effect upon the recursive *sort* procedure for merge-sort.

Consider the definiens of $merge(y_1,y_2,y)$, which has two non-atomic conjuncts D_1 and D_2 :-

$$D_1: \quad (\forall uv)(prec(u,v,y) \leftrightarrow prec(u,v,y_1) \vee prec(u,v,y_2) \vee u \leqslant v, (spans(u,v,y_1,y_2) \vee spans(v,u,y_1,y_2)))$$

$$D_2: \quad (\forall uw)(occurs(u,w,y) \leftrightarrow (\exists w_1 w_2)(occurs(u,w_1,y_1), occurs(u,w_2,y_2), w_1 + w_2 = w))$$

The second of these just ensures that the number of occurrences of any u in y is just the sum of the numbers of occurrences which it has in y_1 and y_2. Clearly D_2 is unaffected by the supposition that members of y_1 are all less than all members of y_2, as they will be if sorted from x_1 and x_2 satisfying $smaller(x_1,x_2)$. However, D_1 is capable of some simplification when this supposition is correct. Observe that D_1 contains the subformula :-

$$u \leqslant v, (spans(u,v,y_1,y_2) \vee spans(v,u,y_1,y_2))$$

If all members of y_1 are less than all members of y_2 then $u \leqslant v$ is implied by $spans(u,v,y_1,y_2)$, so that $(u \leqslant v, spans(u,v,y_1,y_2))$ is S-equivalent to $spans(u,v,y_1,y_2)$. Moreover, the formula $(u \leqslant v, spans(v,u,y_1,y_2))$ is then false, so that the above subformula of D_1 simplifies to $spans(u,v,y_1,y_2)$ when the condition $(perm(x_1,y_1), perm(x_2,y_2), smaller(x_1,x_2))$ holds; this is because that condition trivially implies that all members of y_1 will be less than all members of y_2. The conclusion of this reasoning can be formalized by the S-conditional-equivalence :-

$$(merge(y_1,y_2,y) \leftrightarrow ord(y_1), ord(y_2), D_1',D_2) \leftarrow perm(x_1,y_1), perm(x_2,y_2), smaller(x_1,x_2)$$

where D_1' is the result of simplifying D_1 as just described to :-

$$(\forall uv)(prec(u,v,y) \leftrightarrow prec(u,v,y_1) \vee prec(v,u,y_1,y_2) \vee spans(u,v,y_1,y_2))$$

Now suppose also that y_1 and y_2 are ordered, and append this assumption to the S-conditional equivalence; then the *ord* predicates in the consequent subformula can be replaced by simply *true*, whilst the newly-introduced *ord* antecedents can be partnered with the calls to *perm* to produce calls to *sort* instead. The result of which is :-

$$(merge(y_1,y_2,y) \leftrightarrow D_1',D_2) \leftarrow sort(x_1,y_1), sort(x_2,y_2), smaller(x_1,x_2)$$

Now the subformula (D_1',D_2) exactly describes the relationship between y, y_1 and y_2 which we would customarily write as the predicate $append(y_1,y_2,y)$, and it is easy to show that (D_1',D_2) is S-equivalent to the more usual definiens for *append* (like that used in Section 6.1). Thus that definiens can be replaced here by the *append* predicate to give the sentence :-

$$(merge(y_1,y_2,y) \leftrightarrow append(y_1,y_2,y)) \leftarrow sort(x_1,y_1), sort(x_2,y_2), smaller(x_1,x_2)$$

This lemma makes good sense intuitively. If y_1 and y_2 are ordered

permutations such that all members of y_1 are less than all members of y_2, then merging y_1 and y_2 involves no interleaving and so is equivalent to the appending operation. Observing that the lemma has the familiar form $(F \leftrightarrow F') \leftarrow F''$, we can now make an S-conditional -equivalence for the call to *merge* in the recursive merge-sort procedures for *sort*; the result of which is :-

$$sort(x,y) \leftarrow partition(x_1,x_2,x),\ smaller(x_1,x_2),\ sort(x_1,y_1),\ sort(x_2,y_2),\ append(y_1,y_2,y)$$

This new *sort* procedure asserts the essential logic of quick-sort, which performs all the necessary comparisons between members of x before constructing any ordered permutations.

To obtain a practical quick-sort computation it is necessary to conjointly solve the calls to *partition* and *smaller* in a manner which deals with the partitioning of x deterministically. Hoare's partitioning algorithm chooses any member w from x to leave a non-empty set x', each of whose members is allocated either to a set z_1 (if it is less than w) or else to a set z_2 (if it exceeds w). Then z_1 and z_2 are each quick-sorted to give y_1' and y_2', whence the output y is constructed by appending to y_1' the result of appending y_2' to the unit list (w). This process can be captured in logic by transforming the *sort* procedure above to :-

$$sort(x,y) \leftarrow partition^*(w,x',x),\ allocate(z_1,z_2,x',w),\ sort(z_1,y_1'),\ sort(z_2,y_2'),\ append^*(w,y_2',y'),\ append\ (y_1',y',y)$$

The transformation is conceptually simple but rather laborious, and so is omitted here. The predicate $allocate(z_1,z_2,x',x)$ is specified by :-

$$allocate(z_1,z_2,x',w) \leftrightarrow (\forall u)(\ u \in z_1 \leftrightarrow u \in x',\ u<w\ ),\ (\forall v)(\ u \in z_2 \leftrightarrow u \in x',\ w<v\ )$$

and expresses the fact that w is the discriminator used for allocating the members of x' to either z_1 or z_2 as just described. It is easy to see that x' is properly partitioned by a call to $allocate(z_1,z_2,x',w)$ such that $smaller(x_1,x_2)$ will hold. The sets z_1 and z_2 are related to those named as x_1 and x_2 in the quick-sort *sort* procedure first derived above by $x_1 = \{w\} \cup z_1$, $x_2 = z_2$ (if z_1 is empty) or by $x_1 = z_1$, $x_2 = \{w\} \cup z_2$ (otherwise). These considerations establish that $partition(x_1,x_2,x)$ and $smaller(x_1,x_2)$ are implicitly satisfied by solving a call to *allocate*. Suitable procedures for *allocate* for term representations of sets are trivially derivable, and are just :-

$allocate(\emptyset,\emptyset,\emptyset,w) \leftarrow$

$allocate(u{:}z_1,z_2,u{:}x',w) \leftarrow u<w,\ allocate(z_1,z_2,x',w)$

$allocate(z_1,v{:}z_2,v{:}x',w) \leftarrow w<v,\ allocate(z_1,z_2,x',w)$

Procedures rather similar to these appear in the Clark-Tarnlund paper (16), except that quick-sort is treated there as an algorithm which accepts a list rather than a set as input; it is then necessary to cater for the possibility of identical members in the procedures used in place of those above.

The principal qualities of quick-sort as a 'fast' sorting algorithm depend critically upon the implementation of the partitioning process; the procedures given above only provide a high-level representation of what this partitioning achieves. Hoare's algorithm firstly arranges the input set (or multiset) into a linear array, and then pursues a series of interchanges governed by the bi-directional movement of two pointers. Efficient implementation of this requires a data-overwriting mechanism together with a flag system to control the alternate adjustment of the pointers which indicate the next comparison (and possible interchange). This kind of behaviour could not be feasibly generated from the procedures above with the resources of Prolog ; with a Prolog-like interpreter we would have to devise some more elaborate partitioning procedures which brought the pointer arrangements explicitly into the procedures' argument structures. This is easy enough to accomplish. If the interchanges are somehow implemented upon an internal array representation of the sets, then the subsequent appending operations would no longer be necessary, since the same single array would be adequate to represent both the input and output data; but the latter arrangement is essentially a matter of implementation technology and beyond the scope of the present study.

The Insert-Sort Algorithm

The insert-sort algorithm proceeds by choosing any member w from the input set x to leave the set x', then sorting x' to y', and finally inserting w into the correct position in y' to give the output list y. Clearly this behaviour can be generated from the merge-sort procedure :-

$sort(x,y) \leftarrow partition(x_1,x_2,x),\ sort(x_1,y_1),\ sort(x_2,y_2),$
$merge(y_1,y_2,y)$

by constraining the solution of $partition(x_1,x_2,x)$ so as to compute x_1 as a singleton $\{w\}$. This can be arranged merely by choosing an appropriately restricted (incomplete) procedure set for *partition*. More satisfactorily, the merge-sort procedures can be transformed into a specific program for insert-sort; this will eliminate the need to repeatedly sort singletons by the call to $sort(x_1,y_1)$ in the merge-sort procedure above. The transformation makes use of the elementary relationship :-

$$(partition(x_1,x_2,x) \leftrightarrow partition^*(w,x_2,x)) \leftarrow singleton(x_1,w)$$

in order to make an S-conditional-equivalence substitution for the call to *partition* in the recursive procedure for *sort*, which produces :-

$$sort(x,y) \leftarrow partition^*(w,x',x),\ singleton(x_1,w),\ sort(x_1,y_1),\ [x_2,y_2:=x',y']\quad sort(x',y'),\ merge^*(y_1,y',y)$$

The call $sort(x_1,y_1)$ can be symbolically solved by invoking the merge-sort basis which sorts $\{w\}$ into a unit list (w) ; resolving the basis with the recursive procedure therefore gives :-

$$sort(x,y) \leftarrow partition^*(w,x',x),\ singleton(x_1,w),\ unit\text{-}list(y_1,w),\ sort(x',y'),\ merge^*(y_1,y',y)$$

This procedure clearly performs a constrained merging operation, in that $merge^*(y_1,y',y)$ is called subject to y_1 being just a unit list. But this is the operation which would normally be described as insertion; therefore, to capture that fact in the logic, introduce a new predicate $insert(w,y',y)$ which bears the following relationship to *merge** :-

$$(insert(w,y',y) \leftrightarrow merge^*(y_1,y',y)) \leftarrow unit\text{-}list(y_1,w)$$

Using this to make the obvious S-conditional-equivalence substitution and then deleting the calls $unit\text{-}list(y_1,w)$ and $singleton(x_1,w)$ (since S implies the existences of $\{w\}$ and (w) for any w), the essential *sort* procedure for the insert-sort algorithm is obtained :-

$$sort(x,y) \leftarrow partition^*(w,x',x),\ sort(x',y'),\ insert(w,y',y)$$

The bases for this procedure are just those used in merge-sort.

It is clearly desirable to specialize similarly the procedure set for *merge** to take account of the fact that the first argument in the invoking call can be assumed to be a unit list; the specialized set then serves the new *sort* procedure for insert-sort by virtue of the sentence trivially implied by the relationship between *merge** and *insert* asserted above; that sentence is just the procedure :-

$$insert(w,y',y) \leftarrow unit\text{-}list(y_1,w),\ merge^*(y_1,y',y)$$

Consider one of the *merge** bases :-

$$merge^*(y_1,y',y_1) \leftarrow length(y',0)$$

If the conclusion is substituted by $insert(w,y',y)$ conditional upon y_1 being a unit list, then a basis for *insert* is obtained :-

$$insert(w,y',y) \leftarrow length(y',0),\ unit\text{-}list(y_1,w)$$

The other *merge** basis has no analogous transformation, since it requires y_1 to be the empty list which is clearly inconsistent with an assumption that y_1 is a unit list; hence it does not contribute to the procedure set for *insert*.

Consider next one of the *merge** recursions, making a convenient renaming :-

$$merge^*(y_1,y',y) \leftarrow append^*(w,y_1',y_1),\ append^*(v,y'',y'),\ w \leqslant v,\ merge^*(y_1',y',y^*),\ append^*(w,y^*,y)$$

If y_1 is the unit list (w) then the call $append^*(w,y_1',y_1)$ can be solved immediately to give y_1' as the empty list. In this case the call $merge^*(y_1',y',y^*)$ is also solved immediately by the second *merge** basis which computes $y^*:=y'$. Hence it is easy to see that the assumption that y_1 is a unit list (w) will transform the procedure to a *non-recursive* procedure for *insert* :-

$$insert(w,y',y) \leftarrow append^*(v,y'',y'),\ w \leqslant v,\ append^*(w,y',y)$$

Finally, the other *merge** recursion transforms under the same assumption to give a recursive procedure for *insert* :-

$$insert(w,y',y) \leftarrow append^*(v,y'',y'),\ v \leqslant w,\ insert(w,y'',y^*),\ append^*(v,y^*,y)$$

The logic component of the insert-sort algorithm can now be shown in its entirety, choosing terms as data structures :-

$$sort(\emptyset,nil) \leftarrow$$
$$sort(w{:}\emptyset,w.nil) \leftarrow$$
$$sort(w{:}x',y) \leftarrow sort(x',y'),\ insert(w,y',y)$$

$$insert(w,nil,w.nil) \leftarrow$$
$$insert(w,v.y'',w.v.y'') \leftarrow w \leqslant v$$
$$insert(w,v.y'',v.y^*) \leftarrow v \leqslant w,\ insert(w,y'',y^*)$$

These give a good recursive computation with Prolog-like control.

The Selection-Sort Algorithm

Selection-sort can be viewed as a special case of either quick-sort or insert-sort. The algorithm selects w from x to leave x' such that w is the least member of x; hence x is decomposed *selectively* into $\{w\}$ and x'. After sorting x' to y', the output list y is obtained by appending x' to the unit list (w). Select-sort is thus the special case of insert-sort in which w is always inserted at the beginning of y' to give y, and is the special case of quick-sort where the 'smaller' set x_1 is just a singleton $\{w\}$. Select-sort is derived here by specializing the quick-sort procedures :-

$$\begin{aligned} sort(x,y) &\leftarrow cardin(x,0),\ length(y,0) \\ sort(x,y) &\leftarrow singleton(x,w),\ unit\text{-}list(y,w) \\ sort(x,y) &\leftarrow partition^*(w,x',x),\ allocate(z_1,z_2,x',w), \\ &\qquad sort(z_1,y_1'),\ sort(z_2,y_2'), \\ &\qquad append^*(w,y_2',y'),\ append(y_1',y',y) \end{aligned}$$

Suppose now that w is the least member of x computed by the call to *partition**. Then no members of x are allocated to z_1 by the call to *allocate*. If the *sort* basis is therefore used to solve the first call to *sort* on the assumption that z_1 is the empty set, and if a basis for *append* is likewise invoked to solve the last call to *append*, then the recursive *sort* procedure above simplifies to :-

$$\begin{aligned} sort(x,y) &\leftarrow partition^*(w,x',x),\ allocate(z_1,z_2,x',w), \\ &\qquad cardin(z_1,0),\ sort(z_2,y_2'),\ append^*(w,y_2',y) \end{aligned}$$

However, the properties of *allocate* determine that z_2 and x' must be identical when z_1 is empty. Thus we may write the above as :-

$$\begin{aligned} sort(x,y) &\leftarrow partition^*(w,x',x),\ allocate(\emptyset,x',x',w), \\ &\qquad sort(x',y'),\ append^*(w,y',y) \end{aligned}$$

by the instantiation $z_2{:=}x'$ and macroprocessing out the call to *cardin* for greater conciseness. Now the usual notion of selection can be expressed by a predicate $select(w,x',x)$ which holds when w is selected as the least member from x to leave x'; a fairly intuitive way of specifying this is :-

$$\begin{aligned} select(w,x',x) \leftrightarrow\ & partition^*(w,x',x), \\ & (\forall u)\,(\ w<u \leftarrow u\in x'\ ) \end{aligned}$$

It is then easy to show that the specification can be rewritten :-

$$select(w,x',x) \leftrightarrow (\ singleton(x,w),\ empty(x')\ )$$
$$\vee\ (\ partition^*(w,x',x),\ allocate(\emptyset,x',x',w))$$

by a little case analysis and exploitation of the specification for *allocate*. The consequence of this is that the second basis for *sort* above can be combined with the modified recursive procedure for *sort* just derived, to give a new single *sort* procedure capable of processing any non-empty sets; this procedure is just :-

$$sort(x,y) \leftarrow select(w,x',x),\ sort(x',y'),\ append^*(w,y',y)$$

This and the first *sort* basis comprise a complete procedure set for *sort*.

It is tempting to anticipate that the *allocate* procedures used by quick-sort can be specialized to give useful procedures for selecting the least member w of x. Observe that the specification given for *select* above trivially implies :-

$$select(w,\emptyset,w:\emptyset) \leftarrow$$
$$select(w,x',x) \leftarrow partition^*(w,x',x),\ allocate(\emptyset,x',x',w)$$

Suppose that a new predicate were introduced expressing a special case of *allocate* :-

$$compare(w,x') \leftrightarrow allocate(\emptyset,x',x',w)$$

Only two of the *allocate* procedures shown earlier can deal with cases where the first argument is $\emptyset$; renaming these using the *compare* predicate gives :-

$$compare(w,\emptyset) \leftarrow$$
$$compare(w,v:x') \leftarrow w<v,\ compare(w,x')$$

which could then be used, in principle, to solve the call to *compare* in the paraphrased procedure for *select* :-

$$select(w,x',x) \leftarrow partition^*(w,x',x),\ compare(w,x')$$

In practice, however, these do not constitute an efficient means of achieving the decomposition of x, because the solutions output from the call to *partition** are not constrained. This is reminiscent of the inefficient way of solving the *min* problem which picks a member as a candidate for the minimum and then tests to see if it is a lower bound.

In both problems the remedy is found by exploiting the supposed transitivity of the ordering relation <. In the present case this results in two recursive procedures for *select*, whose effect in execution is to successively discard from *x* those members which cannot be the minimum *w*, meanwhile accumulating these in the other data structure *x'*. These procedures are shown below together with the rest of the complete program body :-

sort(∅,nil) ←
sort(x,w.y') ← *select(w,x',x)*, *sort(x',y')*

select(w,∅,w:∅) ←
select(w,v:x",u:v:x") ← *u<v*, *select(w,x",u:x")*
select(w,v:x",v:u:x") ← *u<v*, *select(w,x",u:x")*

These generate a satisfactory iterative computation from a typical logic program interpreter.

7.2 : LOGIC PROGRAMS FOR STRING SEARCHING

The String Searching Problem

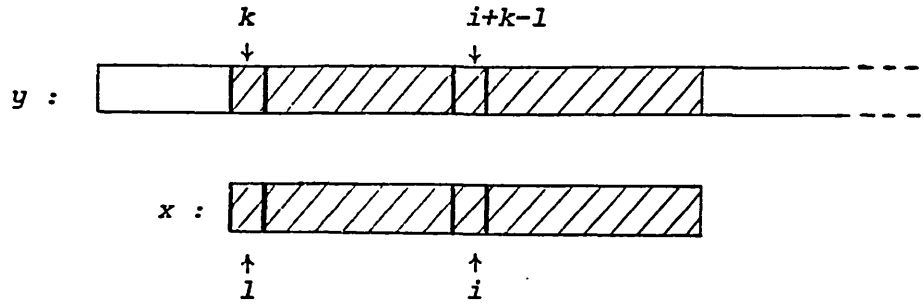
The problem considered in this final section arises in the general field of text processing, which encompasses a rich class of problems concerning the analysis of symbol strings. The present study examines the specific task of determining whether a given string (the 'keyword') occurs in some other string (the 'text string'). This task is obviously of paramount importance in applications such as text editing and bibliographic retrieval. In applications such as those, much can be gained in terms of computational efficiency by employing techniques such as indexing, hash-addressing and so forth to allow rapid retrieval; these generally proceed by consulting other elaborate data structures established by pre-processing the keyword or the text string. The objective of such techniques is to improve upon the simplest algorithm which conducts a sequential search through the text string, potentially inspecting all its symbols. Here we derive the logic representation of this 'naive' algorithm, and then consider how it may be transformed into two somewhat more intelligent algorithms which refer to pre-processed data structures in order to restrict the search without missing potential solutions.

The problem can be formulated in logic by introducing the predicate *string(x,y)* which holds when the keyword *x* has an occurrence in the text string *y*; more briefly, we can say that *x* is a string in *y* when the predicate holds. In order to specify this predicate precisely it is convenient to summon the *item* predicate and so express the relationship between *x* and *y* in terms of their indexed members. Assuming also the axioms *A1-A3* used elsewhere to constrain the data types possessing indexed members, the principal specification can be announced as :-

$$string(x,y) \leftrightarrow (\exists k)(\forall ui)(item(u,i+k-1,y) \leftarrow item(u,i,x))$$

which simply requires that there is some k^{th} member of *y* with which the first member, if any, of *x* can be aligned such that all members of *x* then match their counterparts in *y*. This circumstance is

depicted in the diagram below in which the shaded sections are matching symbol strings :-



Observe that the given specification allows the possibility that x is the empty string, in which case it is a string in y with k undetermined. It should also be noted that the *string* relation is transitive, as is usually the case with relations having definiens' of the form $(A \leftarrow B)$. In fact it can easily be ascertained that the following slightly stronger property of *string* also holds :-

$$string(x,y) \leftrightarrow (\exists z)(string(z,y),\ string(x,z))$$

The computational significance of this property is that in order to show that x is a string in y, it is sufficient to find a string z in y in which x is a string. All the algorithms considered here exploit this property in one way or another.

The Naive (Quadratic) Algorithm

The naive algorithm attempts to find x in y by sequentially inspecting the members of y until discovering one which matches the first member of x; having thus found a potential solution, the algorithm pursues a process of comparing the members of x with those to which they align in y when the first two matching members are aligned. In other words, having found a potential solution beginning at the k^{th} member in y as in the diagram above, a local matching exercise is conducted to discover whether the shaded sections shown there can be matched member for member. If the latter process encounters a mismatch, then the original search is resumed from its current point to seek a new k^{th} member in y which matches the first member in x.

In all the logic representations of the algorithms examined here it is convenient to make use of the notion of a *prefix*. A

prefix x of a string z is some string in z whose first member, if any, coincides with the first member of z. Another way of expressing this is to say that every k^{th} member of x is identical to the k^{th} member of z. This can be captured by the following specification of *prefix(x,z)* which holds when x is a prefix of z :-

$$prefix(x,z) \leftrightarrow (\forall ui)(item(u,i,z) \leftarrow item(u,i,x))$$

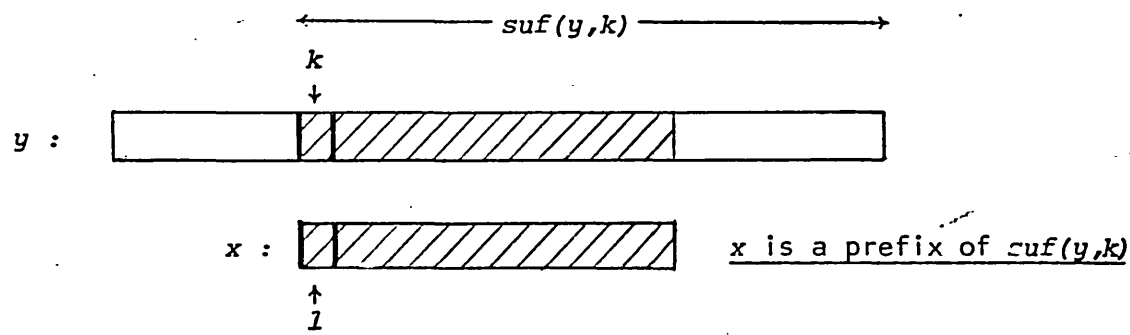
Observe that this admits the possibility that x can be the empty string and a prefix of (any) z.

Another useful notion is that of *suffix*. A suffix z of y is a string in y such that the last members of z and y coincide. This can be expressed by identifying z with a string in y denoted by the term *suf(y,k)*; this term denotes the string in y which extends from its k^{th} member right up to its end. The relationship between the indexed members in y and those in its k^{th} suffix *suf(y,k)* is expressed by the following axiom in S :-

$$item(u,i-k+1,suf(y,k)) \leftrightarrow item(u,i,y),\ i \geqslant k$$

Hence our logical treatment of the string searching problem will refer to prefixes using the *prefix* predicate, but to suffixes using the *suf* term.

These definitions now allow a useful way of viewing the naive algorithm : that algorithm successively inspects the suffixes *suf(y,1)*, *suf(y,2)*, ..., *etc.* seeking some suffix *suf(y,k)* of which x is a prefix. This is depicted in the diagram below :-



Now it is possible to see how the naive algorithm exploits the transitivity of the *string* relation. Instantiate the transitivity axiom :-

$$string(x,y) \leftarrow string(z,y),\ string(x,z)$$

with the choice $z:=suf(y,k)$. The *string* specification together with

the axiom specifying indexed membership in suffixes just given jointly imply :-

$$string(suf(y,k),y) \leftarrow$$
$$string(x,suf(y,k)) \leftarrow prefix(x,suf(y,k))$$

Invoking these in response to the two 'calls' in the instantiated transitivity axiom then produces :-

$$string(x,y) \leftarrow prefix(x,suf(y,k))$$

This sentence may be viewed as a procedure which selects some k^{th} suffix of y and tests whether it has x as a prefix; if not, some other k must be tried. Given a procedure set solving calls to *prefix*, the *string* procedure above would be sufficient - that is, would be a complete procedure set for *string*. This fact can be shown by the proof :-

$$\vdash_S string(x,y) \leftrightarrow (\exists k)(\forall ui)(item(u,i+k-1,y) \leftarrow item(u,i,x))$$
$$\vdash_S string(x,y) \leftrightarrow (\exists k)(\forall ui)(item(u,i,suf(y,k)) \leftarrow item(u,i,x))$$
$$\vdash_S string(x,y) \leftrightarrow (\exists k)prefix(x,suf(y,k))$$

This guarantees that there must exist some suffix $suf(y,k)$ of which x is a prefix when $string(x,y)$ is solvable. This is a very important fact about the problem domain, and is the basis of other algorithms as well as the naive one. Note that it embodies two computational concepts : <u>*suffix selection*</u>, which consists of choosing some k^{th} suffix of y in which to seek the string x; and <u>*prefix testing*</u>, which consists of determining whether x is a prefix of some string. The naive algorithm iteratively selects suffixes, each time applying the prefix test, and it potentially investigates every suffix $suf(y,k)$ for $k = 1,2, \ldots$, etc. in that order. There are a number of alternative representations of the logic component of the naive algorithm, which differ in the amount of information about the progress of computation that they encode within their argument structures. We shall develop these in order of increasing information in that respect, and so arrive at a logic component which is sufficiently informed to allow some useful transformations leading to more sophisticated algorithms.

1] The Simplest Representation

The first representation of the naive algorithm considered here makes use of no other information than the *string* specification together with some axioms about data constructibility. It is assumed that strings are either empty strings or else constructible from a call *append*(u',z',z)* which holds when string *z* has first member *u'* and the rest of the string *z* is *z'*:-

$$empty\text{-}list(z) \leftrightarrow (\forall ui)(item(u,i,z) \leftrightarrow false)$$

$$append^*(u',z',z) \leftrightarrow (\forall ui)(item(u,i,z) \leftrightarrow item(u,i-1,z') \vee (u=u',i=1))$$

Admitting these to the specification set allows the derivation of procedures for *string* to be trivially pursued just as with many other examples given in the thesis; there is no need to present the derivations in detail. By considering the two cases *k=1* and *k>1* it is easy to produce the procedure set :-

$$string(x,y) \leftarrow prefix(x,y) \qquad [k=1]$$

$$string(x,y) \leftarrow append^*(v',y',y),\ string(x,y') \qquad [k>1]$$

Jointly these say that to show that *x* is a string in *y*, either show that *x* is a prefix of the first suffix of *y*, or else show that it is a prefix of one of the remaining suffixes - that is, a string in the rest of *y*. A procedure set for *prefix* is obtained by just considering the cases of the construction of *x* and *y*. It is easy to show that all cases are dealt with by just two procedures :-

$$prefix(x,y) \leftarrow empty\text{-}list(x)$$

$$prefix(x,y) \leftarrow append^*(u',x',x),\ append^*(u',y',y),\ prefix(x',y')$$

The first of these deals with all cases of *y* when *x* is empty. The second one deals with all cases where both *x* and *y* are constructible by *append**; there is no procedure for the case where *x* is so constructible but *y* is not, because it can be shown that the *prefix* problem is not then solvable. Perhaps it should also be noted that there is no need to compose a procedure for *string* for the case where *k>1* but *y* is not constructible by *append** - this could only be solved with *x* as the empty string, and the first *string* procedure can deal with this by invoking the *prefix* basis. The computation typically generated from the *prefix* procedures is just a fast iteration which repeatedly accesses aligned members in *x* and *y* and compares them.

The procedures above can be made very concise using terms to represent the strings; by introducing these through the device of macroprocessing we obtain a representation of the naive algorithm which appears to be simpler than all others :-

string(x,y) ← *prefix(x,y)*
string(x,v'.y') ← *string(x,y')*

prefix(nil,y) ←
prefix(u'.x',u'.y') ← *prefix(x',y')*

The computation generated from these procedures exhibits non-determinism by virtue of the two ways of responding to a call to *string*. Clearly the second procedure for *string* can be repeatedly invoked to select any suffix before applying any prefix test. However, a Prolog-like execution applied to the procedures as scheduled in the order given above would always defer the generation of the next suffix until completing the prefix test upon the currently inspected suffix. This is the most sensible schedule to use in the absence of any information about the likeliest region of y, if any, in which x may appear as a prefix. If the scheduling of the two *string* procedures is the reverse of that just suggested, the prefix test is applied instead to suffixes of y in order of decreasing k, which will not come about until the computation from the second procedure has iterated right through y to arrive at the empty string - at which point a stack of suffixes is represented in the binding environment, each one awaiting its prefix test.

A simple way of visualizing the effects of these schedules is to imagine that the keyword x 'slides' one position alongside the text string y each time a new suffix is selected. The former schedule slides x from left-to-right (treating y's first member as left-most), attempting at each new alignment to match x with the substring of y with which it is contiguous. The other schedule slides x from right-to-left, which is a reasonable strategy when there is reason to believe that x occurs in y near its right-end; however, the procedures above do not implement that strategy effectively because they do not give direct access to the right-end part of y.

The order of suffix selection can be enforced by the logic by making a simple modification to the second *string* procedure. Suppose that the first members of x and y are distinct. Then x cannot be a prefix of y, and so *string(x,y)* can only be solved by

showing that x is a string in the rest of y. Therefore replace the second procedure by :-

string(u'.x', v'.y') ← u'≠v', string(v'.x',y')

The computation is now much more deterministic, because the control call $u'\neq v'$ suppresses the possibility of selecting a new suffix before completing the prefix test for the current one; execution successively applies the prefix test to *suf(y,1), suf(y,2), ..., etc.* as x slides from left-to-right alongside y. *[*It is assumed throughout that the control is Prolog-like.*]* Observe that in the original *string* procedure set the tasks of suffix selection and prefix testing were initiated by distinct *string* procedures, whereas with the new arrangement the prefix test is shared between the two *string* procedures. We can regard the call $u'\neq v'$ as a device encoding control information about the potential failure of a call to the other *string* procedure; that is, the second procedure is effectively informing the computation that the first one cannot succeed when the first members of x and y are distinct.

The naive algorithm is so-called, not because of its central features of suffix selection and prefix testing, but rather because it potentially selects *every* suffix; using the sliding notion, the characteristic feature of the naive algorithm's exhaustive suffix selection is that when a mismatch occurs during a prefix test, the keyword x slides just *one* position down y , which is tantamount to selecting the next suffix (increasing k by 1). More sophisticated algorithms permit x to slide several positions after a mismatch before the resumption of comparing members, and are consequently able to make fewer comparisons than does the naive algorithm. The ability to slide x several positions without any intermediate member comparisons depends upon knowledge about the instigating mismatch in order to ensure that no solutions are 'skipped over' as x slides past intermediate positions. Knuth, Morris and Pratt (44) have shown that the worst-case behaviour of the naive algorithm gives a comparison count approaching $\hat{k}.L$ where $\hat{k}$ is the least suffix pointer for which *prefix(x,suf(y,$\hat{k}$))* holds and L is the length of x. Now it can be argued that the average penetrance $\hat{k}$ over all keywords of some length L for a given text string is a monotonically increasing function of L, and so this together with the result above shows that $\hat{k}$ has a non-linear dependence upon L; in fact in the worst case that dependence

is almost quadratic in L; this is the case when y contains many proper substrings of x with length approaching L. An extreme example is where y has the form $a^m b$ and x has the form $a^n b$ with $m >> n$. In realistic circumstances, such as natural language text searching with large phrases as keywords, y will contain relatively few large substrings of x and the comparison count will approximate more closely to $\hat{k}$ than to $\hat{k}.L$, so that the algorithm's efficiency is then approximately linear in L.

As already mentioned, the more elaborate algorithms which are available for the string searching problem have the ability to decide how to re-align x after a mismatch on the basis of an analysis of the context in which that mismatch occurred. The more penetrating that analysis of the failure to solve *prefix(x,y)* , the more intelligent can be the subsequent choice of suffix. In order to express such analyses in the logic representation of an algorithm of this kind it is necessary to arrange that enough information is held in the procedures' argument structures for the context of the mismatch to be ascertained. The minimal information in this respect is the position k in y with which the first member of x currently aligns. Additionally it would be useful to have direct access to the positions of the mismatch in question. In the following discussions, further representations of the naive algorithm are developed which arrange for this kind of information to be represented explicitly, rather than implicitly in the current binding environment.

2] *Explicit Control of Suffix Selection*

The logic of the naive algorithm can be slightly elaborated in a way which makes little improvement upon its efficiency but which is nevertheless very instructive for our pursuit of logic representations of better algorithms. Consider the behaviour of the current program when a mismatch occurs during a *prefix* computation. When this occurs the interpreter has to backtrack in order to find out how to choose the next suffix. By backtracking to the activation of the call which invoked the first *string* procedure, and thereby instigated this particular prefix test, the interpreter discovers, in effect, the identity of the current suffix as represented by the second argument of that call; then, by transmitting that argument during the invocation of the alternative *string* procedure in response to that call's re-activation, the latter procedure becomes informed about the

identity of the current suffix and so is able to generate the next one. It is interesting to secure these arrangements for responding to a failed prefix test by expressing them in the logic component; this requires the construction of somewhat more elaborate predicates than those introduced so far.

Consider a predicate *string*(x,y,w,z)* which has the meaning that either *x* is a prefix of *y* or else *w* is a string in *z* :-

$$string^*(x,y,w,z) \leftrightarrow prefix(x,y) \vee string(w,z)$$

This anticipates the run-time circumstance when a test is initiated to find out whether some *x* is a prefix of some *y*; if the test fails, then computation assumes that the only way remaining in which to solve the original goal $\leftarrow string(\hat{x},\hat{y})$ is to show that *w* is a string in *z*. We shall see presently that it is possible to compose a program body using the new predicate which arranges that when a prefix test is initiated it is supplied with a record of the next suffix which must be inspected if the test fails; that record is maintained in a directly accessible state in the last two argument positions of the procedures which will be used to conduct the test, so that no backtracking is needed for their retrieval. This is just another instance of how suitable choices of programming style can allow the logical representation of matters which would otherwise be treated as control information.

Recalling the procedure set already established for strings represented by terms, the completeness of the procedure set for *string* establishes the sentence :-

$$string(x,v'.y') \leftrightarrow prefix(x,v'.y') \vee string(x,y')$$

However, the disjunction in this sentence unifies with the definiens given above for *string**, so that the following holds :-

$$string(x,v'.y') \leftrightarrow string^*(x,v'.y',x,y')$$

This shows that a call to *string* can be investigated using the procedure :-

$$string(x,v'.y') \leftarrow string^*(x,v'.y',x,y')$$

together with procedures solving the call to *string**. Observe, then, that when we wish to solve a goal $\leftarrow string(\hat{x},\hat{y})$ in this way, the call to *string** will associate its first two arguments with the task

of showing that $\hat{x}$ is a string in the first suffix of $\hat{y}$ (by showing that it is a prefix of $\hat{y}$), whilst its last two arguments are associated with the task of showing that $\hat{x}$ is a string in one of the remaining suffixes of $\hat{y}$ (by showing that $\hat{x}$ is a string in *suf(y,2)*).

A procedure set for *string** can be derived easily using the knowledge already available. The completeness of the set of *prefix* procedures already given establishes the sentence :-

$$prefix(u'.x',\ u'.y') \leftrightarrow prefix(x',y')$$

so that the first disjunct in the definiens of *string** may be replaced accordingly with an *S*-equivalent formula giving :-

$$string^*(u'.x',u'.y',w,z) \leftrightarrow prefix(x',y') \vee string(w,z)$$

We may infer a procedure for *string** from this as follows :-

$$string^*(u'.x',u'.y',w,z) \leftarrow string^*(x',y',w,z)$$

The completeness of the procedure set for *prefix* also determines that *prefix(u'.x',v'.y')* is *false* when u' and v' are distinct, which easily provides another procedure for *string** by virtue of the following deductions :-

$$\vdash_S string^*(u'.x',v'.y',w,z) \leftrightarrow prefix(u'.x',v'.y') \vee string(w,z)$$
$$\vdash_S (string^*(u'.x',v'.y',w,z) \leftrightarrow false \vee string(w,z)\ ) \leftarrow u' \neq v'$$
$$\vdash_S string^*(u'.x',v'.y',w,z) \leftarrow u' \neq v',\ string(w,z)$$

The *string** synthesis has so far considered the two cases in which $\hat{x}$ is non-empty and the first member of $\hat{x}$ either does or does not match the first member of $\hat{y}$. When $\hat{x}$ is empty the definiens of *string** is made *true* because the *prefix* basis shows that empty $\hat{x}$ is a prefix of any $\hat{y}$, from which we infer a basis for *string** :-

$$string^*(nil,y,w,z) \leftarrow$$

Finally there is the possibility that a call is made to *string* with $\hat{y}$ empty; this cannot invoke the procedure already inferred above which investigates *string* by investigating *string**. However, when $\hat{y}$ is empty then the call can only be solved when $\hat{x}$ is also empty, and so this final case amongst those cases of the input strings is dealt with by the *string* basis :-

$$string(nil,nil) \leftarrow$$

Clearly all these procedures could be derived very straightforwardly in the goal-directed format of previous examples, requiring only trivial subformula substitutions and simplifications. Their behaviour is quite interesting, and so it is worth gathering them all together for further contemplation :-

$$string(nil,nil) \leftarrow$$
$$string(x,v'.y') \leftarrow string^*(x,v'.y',x,y')$$

$$string^*(nil,y,w,z) \leftarrow$$
$$string^*(u'.x',u'.y',w,z) \leftarrow string^*(x',y',w,z)$$
$$string^*(u',x',v'.y',w,z) \leftarrow u' \neq v', string(w,z)$$

Consider a call $string(\hat{x},\hat{y})$ for instances of $\hat{x}$ and $\hat{y}$ in the case where $\hat{y}$ is not empty. By invoking the second *string* procedure, a record $w:=\hat{x}$, $z:=\hat{y}'$ is established in the last two arguments of the *string** call which effectively describes the way in which the original problem might yet be solved if subsequent computation fails to show that $\hat{x}$ is a prefix of $\hat{y}$. During the prefix test, which is conducted by the first two *string** procedures, this record is preserved in readiness for a failure due to mismatch; in such an event, the third *string** procedure accesses this record and injects it into a new computation which has the object of solving $string(w,z)$, this being the only remaining way in which the original call $string(\hat{x},\hat{y})$ can be solved. Execution is now very deterministic, instigating very little backtracking. This does not necessarily improve upon the run-time behaviour of the naive algorithm when using the simpler logic component instead, because a modest interpreter ought to be able to manage the latter's backtracking quite efficiently. But we are more concerned at present with the information encoded by the procedures rather than with their behavioural attributes.

3] *Explicit Control of Comparison Positions*

In order to analyse fully a failure to match some member of x with a member of y it is necessary to know the positions at which these members occur in the respective strings. These positions are not known within the logic representations considered so far. For example, consider the computation instigated by the goal :-

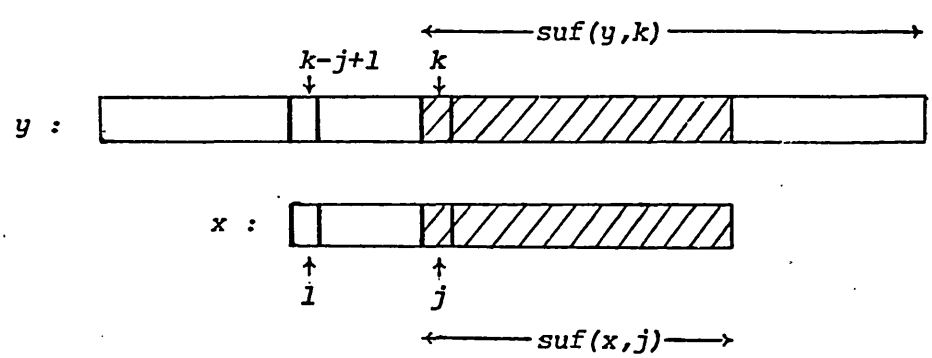
← *string(a.b.c.nil , b.a.b.a.b.a.b.c.nil)*

by the first of the procedure sets already examined. Eventually a failure is encountered in the attempt to show that *a.b.c.nil* is a prefix of *a.b.a.b.c.nil* ; this failure is signalled by the mismatch of the member *c* in $\hat{x}$ with the third occurrence of *a* in $\hat{y}$ ($\hat{x}$ and $\hat{y}$ being the original arguments in the goal shown above). The goal at the point in the computation when the mismatch occurs is :-

← *prefix(c.nil , a.b.c.nil)*

which contains no information about the positions in $\hat{x}$ and $\hat{y}$ of the mismatched members. Of course, that information could be recovered from the run-time stack by determining various counts of procedure invocations, but the point at issue here is that the positions of the mismatched members are not represented explicitly in any way by which our procedures may directly refer to them. Hence those procedures cannot express in logic the relationship between those positions and the best way to proceed with the remaining task of showing that *a.b.c.nil* is a string in *a.b.a.b.c.nil* now that it is known not to be a prefix of *a.b.a.b.c.nil*.

To provide for the analysis of mismatch positions it is necessary to introduce a new predicate which reflects the general view of the string searching algorithm as a controller of two pointers *j* and *k* respectively pointing to the members of *x* and *y* next to be compared; the adjustments of these pointers can be interpreted in terms of suffix selection and prefix testing. Let the predicate *prefix*(x,y,j,k)* hold when the string *suf(x,j)* is a prefix of the string *suf(y,k)* ; this is depicted in the diagram below, where the shaded regions signify the members which *remain to be matched* in order to solve the call *string(x,y)* :-



shaded regions match when *prefix*(x,y,j,k)* holds

The logical expression of this new predicate can be expressed either in terms of the indexed members of x and y :-

$$prefix^*(x,y,j,k) \leftrightarrow (\forall ui)(item(u,k-j+1,y) \leftarrow item(u,i,x), i \geq j)$$

or else, at a somewhat higher level, using the suffix notation :-

$$prefix^*(x,y,j,k) \leftrightarrow prefix(suf(x,j), suf(y,k))$$

These specifications are S-equivalent by virtue of the axiom given previously defining the meaning of indexed membership for suffixes constructed from *suf*. Now consider the case where j takes the value 1; then we have :-

$$\vdash_S \quad prefix^*(x,y,1,k) \leftrightarrow (\forall ui)(item(u,k-1+i) \leftarrow item(u,i,x), i \geq 1)$$
$$\vdash_S \quad prefix^*(x,y,1,k) \leftrightarrow (\forall ui)(item(u,k-1+i) \leftarrow item(u,i,x))$$
$$\vdash_S \quad (\exists k)\, prefix^*(x,y,1,k) \leftrightarrow string(x,y)$$

This establishes that $string(x,y)$ can be investigated completely by calling $prefix^*(x,y,1,k)$. Procedures capable of dealing with such a call can be derived very easily, and are closely analogous to those given previously for investigating $prefix(x,y)$. They are :-

$$prefix^*(x,y,j,k) \leftarrow length(x,w),\ j>w$$
$$prefix^*(x,y,j,k) \leftarrow item(u,j,x),\ item(u,k,y),\ prefix^*(x,y,j+1,k+1)$$

The recursive *prefix** procedure tries to show that $suf(x,j)$ is a prefix of $suf(y,k)$ by showing that both strings have the same first member u and that $suf(x,j+1)$ is a prefix of $suf(y,k+1)$. The basis procedure deals with the case where $suf(x,j)$ is empty, this being so when j exceeds the length w of x. The suffixes referred to here are, of course, represented only implicitly by the *prefix** procedures through the pointers j and k. They essentially paraphrase the *prefix* procedures shown below :-

$$prefix(suf(x,j), suf(y,k)) \leftarrow length(suf(x,j), 0)$$
$$prefix(suf(x,j), suf(y,k)) \leftarrow append^*(u,suf(x,j+1),suf(x,j)),\ append^*(u,suf(y,k+1),suf(y,k)),\ prefix(suf(x,j+1), suf(y,k+1))$$

which result by simply instantiating the general procedures for $prefix(\tilde{x},\tilde{y})$ with $\tilde{x}:=suf(x,j)$, $\tilde{y}:=suf(y,k)$.

A complete program body for the string searching problem can now be composed of just the initiating procedure :-

string(x,y) ← *prefix*(x,y,1,k)*

together with the two procedures given for *prefix** and some means of accessing the indexed members of *x* and *y*. In conjunction with a goal ← *string(x,y)* they just contribute a new logic component for the naive algorithm. It is interesting to note that whereas the previous procedure set for *string* :-

string(x,y) ← *prefix(x,y)*
string(x,u'.y') ← *string(x,y')*

gave rise to non-determinism through offering two alternative ways of responding to a call to *string* the new body's non-determinism arises through the non-deterministic solution of the call to *item*. A typical computation from the new procedures responds to a failure to solve some call *prefix*(x,y,j,k)* by backtracking to the most recently activated call *item(u,k,y)* which was activated with *k* as an output argument in order to solve *prefix*(x,y,1,k)* ; since the choice of *k* computed by that call has resulted in a failure on the prefix test, the call must be re-activated to seek an alternative choice of *k*. The response to the call *item(u,k,y)* obviously depends upon the arrangements made for interrogating the string *y*; when the members of *y* are accessed serially, such as by applying Prolog-like control to the procedures below :-

item(u',1,u'.y') ←
item(u,k,u'.y') ← *item(u,k-1,y')*

then the computation behaves in the manner already described for the simplest representation given earlier for the naive algorithm. The non-deterministic selection of *k* by the call *item(u,k,y)* in the new procedures corresponds to the non-determinism manifested in the two *string* procedures in the simplest representation in that the second of them can be recursively invoked arbitrarily many times before the first one is invoked ; each of those invocations of the second *string* procedure implicitly selects a new k^{th} member of *y* with which the first member of *x* is aligned.

In the next, and final, logic representation of the naive algorithm, the ideas of the two former representations are combined.

4] *Explicit Suffix Selection Using Pointers*

By holding two pointers j and k in the argument structure, it is possible to control the selection of suffixes deterministically through the agency of the logic component using much the same idea as employed in the earlier representation *2]*. This requires the introduction of a new predicate *string**(x,y,j,k)* which holds when either *suf(x,j)* is a prefix of *suf(y,k)* or else x is a string in *suf(y,k-j+2)*. The specification is therefore :-

*string**(x,y,j,k)* ↔ *prefix*(x,y,j,k)* ∨ *string(x,suf(y,k-j+2))*

An informal explanation of this choice of predicate is as follows : suppose that computation has proceeded to the point where the j^{th} member of x is aligned with the k^{th} member of y in the course of investigating the call *string(x,y)* ; assume that the preceding members of x, if any, have been matched with their counterparts in y *[note, with care, that the string** specification above does not insist upon this, but only considers matches at and beyond the* j^{th}*]* ; the problem is then solved either by matching the members at and beyond the pointers j and k, or else a new suffix must be selected; the current suffix is *suf(y,k-j+1)* with the supposed alignment of the j^{th} and k^{th} members of x and y respectively; the naive algorithm chooses the next suffix as *suf(y,k-j+2)* in which to show that x is a string.

A program body using the *string*** predicate can be derived exactly as for the previous representation in which comparison pointers were only implicit. It turns out to be :-

```
string(x,y) ← string**(x,y,1,1)

string**(x,y,j,k) ← length(x,w), j>w
string**(x,y,j,k) ← item(u,j,x),
                    item(u,k,y), string**(x,y,j+1,k+1)
string**(x,y,j,k) ← item(u,j,x),
                    item(v,k,y),
                          u≠v, string**(x,y,1,k-j+2)
```

The first two *string*** procedures behave rather like the *prefix** procedures, except that k is now an input argument in every *string*** invocation. When a mismatch occurs, directing control to the third *string*** procedure, this procedure uses the directly accessible pointers j and k of the mismatched members in order to determine the

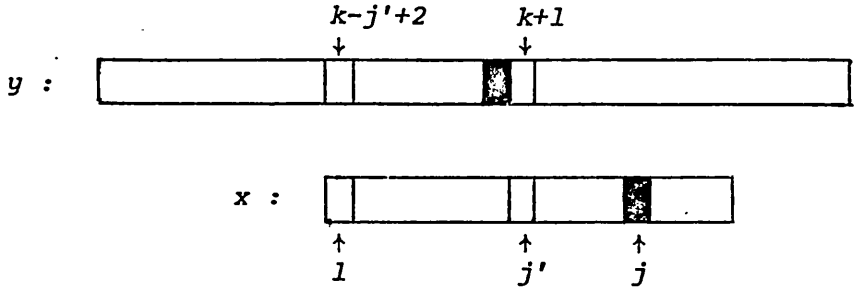
identity of the next suffix; since this, in the naive algorithm, is just *suf(y,k-j+2)*, the values of j and k comprise sufficient information within the procedures' argument structures for this determination to be made. There is therefore no significant backtracking during the computation. Observe also that, because the arguments j and k in the calls to *item* are always input instances, the solution of those calls is deterministic; this allows the recursive *string*** procedures to be invoked iteratively, in contrast to the necessarily recursive invocation of the *string** procedures in the previous representation.

The procedures above using the *string*** predicate provide a very satisfactory account of the logic which underlies the naive algorithm. More importantly, because they introduce to the logic explicit arrangements for accessing the mismatch position - and hence the current suffix identity - they form an excellent basis for deriving the logic components of those algorithms which use that information to decide whether the keyword can slide more than one position after each mismatch without missing potential solutions. These are the algorithms considered next.

The Linear Algorithm

In order to introduce the linear algorithm, which is due to Knuth, Morris and Pratt (44), it is useful to consider the behaviour of the *string*** program examined above. Suppose that a call to *string**(x,y,j,k)* is executed with $x_j \neq y_k$. The ensuing mismatch signals the fact that x is not a prefix of the current suffix *suf(y,k-j+1)*. Thus to show that x is a string in *suf(y,k-j+1)* it must be shown that x is a string in *suf(y,k-j+2)*. The third *string*** procedure above investigates this latter objective by displacing x by just *one* position relative to its current alignment with y, so that whereas x_1 was formerly aligned with y_{k-j+1} when the mismatch occurred, the displacement now aligns x_1 with y_{k-j+2}. A consequence of displacing x by one position is that, depending upon the nature of x and y, the member y_k may subsequently be compared with x_{j-1} in the course of showing x to be a prefix of *suf(y,k-j+2)*, in which event the computation compares y_k with a member of x more than once. It is for this reason that the naive algorithm gives behaviour which is generally worse-than-linearly dependent upon the length of the keyword.

Suppose that when the mismatch occurs between x_j and y_k, x is immediately displaced such that x_1 aligns with some $y_{k-j'+2}$ where $j' \leq j$. Then y_k aligns with $x_{j'-1}$, and y_{k+1} aligns with $x_{j'}$. This is shown below (that is, after the displacement) :-



next alignment after mismatch of x_j and y_k (shaded)

Observe that after the mismatch x has been displaced by $j-j'+1$ positions; the naive algorithm always chooses $j=j'$ whereas we are now considering algorithms which choose $j'<j$ and thus displace x by several positions.

The linear algorithm has the characteristic property that when x_j does not match with y_k, the values of j and k are used to compute a particular value of j' such that by displacing x by $j-j'+1$ positions the comparison of members can be resumed starting with $x_{j'}$ and y_{k+1}. This arrangement assumes, firstly, that the determination of j' is such that any members of x preceding $x_{j'}$ will match those members of y with which they align after the displacement (so that there is no need to match them again), and assumes secondly that no opportunities for solving *string(x,y)* are missed by displacing x by more than one position when $j'<j$. When the displacement of x after each mismatch satisfies these conditions, no member of y is ever compared more than once; the resulting algorithm's behaviour is therefore linear in $\hat{k}$ and thus (because of the argument about the mean penetrance) linear in the length L of the keyword. Observe, however, that when the given goal is solved with x_1 finally aligned with $y_{\hat{k}}$, exactly $\hat{k}+L-1$ of the members of y will have been compared with members of x; later on we consider an algorithm which improves upon this.

The logic of the linear algorithm can be derived by modifying the third *string*** procedure given previously, since this is the procedure responsible for selecting a new suffix after a mismatch.

We require that instead of calling *string**(x,y,1,k-j+2)*, that procedure should compute an appropriate *j'* and then call *string**(x,y,j',k+1)*, and that this modification will not allow any solutions to be missed; in other words, we require a more efficient, but nevertheless complete, procedure set for *string***. The precondition for such a modification is apparent from consideration of the circumstances when, generally, $string(x,suf(y,k_1))$ and $string(x,suf(y,k_2))$ are *S*-equivalent when $k_1 \leqslant k_2$. Using the *string* specification it is easy to prove that this *S*-equivalence holds when *x* is not a prefix of any suffix *suf(y,k*)* satisfying $k_1 \leqslant k^* < k_2$. Applied to the present context, this fact is expressible by the *S*-conditional-equivalence :-

$$(string(x,suf(y,k-j+2)) \leftrightarrow string(x,suf(y,k-j'+2)))$$
$$\leftarrow j' \leqslant j,\ \sim(\exists k^*)(k-j+2 \leqslant k^* < k-j'+2,\ prefix(x,suf(y,k^*)))$$

Informally, this just states the fairly obvious fact that, given the goal of showing that *x* is a string in *suf(y,k-j+2)*, which is the naive algorithm's goal after the mismatch of x_j and y_k, it is possible instead to just determine whether *x* is a string in the suffix *(j-j')* positions further on, provided that $j' \leqslant j$ and there is no intermediate suffix amongst those ignored in which *x* could be a prefix. This lemma allows an *S*-conditional-equivalence substitution in the third *string*** procedure when the latter is written as :-

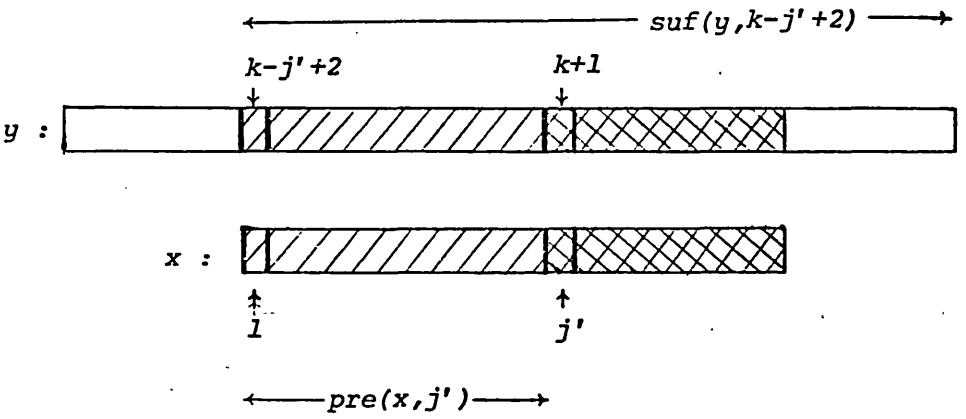
$$string^{**}(x,y,j,k) \leftarrow item(u,j,x),\ item(v,k,y),\ u \neq v,\ string(x,suf(y,k-j+2))$$

by replacing its original call *string**(x,y,1,k-j+2)* by the *S*-equivalent call *string(x,suf(y,k-j+2))*. The result of this inference is the procedure :-

$$string^{**}(x,y,j,k) \leftarrow item(u,j,x),\ item(v,k,y),\ u \neq v,\ j' \leqslant j,$$
$$\sim(\exists k^*)(k-j+2 \leqslant k^* < k-j'+2, prefix(x,suf(y,k^*))),$$
$$string(x,suf(y,k-j'+2)).$$

Having modified the procedure in this way, the next objective is to accommodate in the logic the requirement mentioned earlier that comparisons are to be resumed beginning with $x_{j'}$ and y_{k+1}. This suggests that there must be some way of introducing a call *string**(x,y,j',k+1)* to the modified procedure which will replace

the rather less useful call to *string* introduced by the lemma. In fact there is quite an easy substitution which satisfies this objective and which comes about from considering the position as depicted below after the displacement of *x* has taken place :-



when single-shaded-only sections match,

prefix(pre(x,j'), suf(y,k-j'+2)) holds

when double-shaded-only sections match,

*string**(x,y,j',k+1)* holds

when both shaded sections match,

string(x,suf(y,k-j'+2)) holds

A term *pre(x,j')* has been introduced in order to refer conveniently to the prefix of *x* which extends up to the member, if any, which immediately precedes the $(j')^{th}$. With this arrangement, the matching in the picture above can be expressed by the obvious lemma below, which just expresses the matching of *x* with its contiguous section in *y* in terms of matching a prefix of *x* and a suffix of *x* :-

string(x,suf(y,k-j'+2)) ↔ *prefix(pre(x,j'), suf(y,k-j'+2)),*
*string**(x,y,j',k+1)*

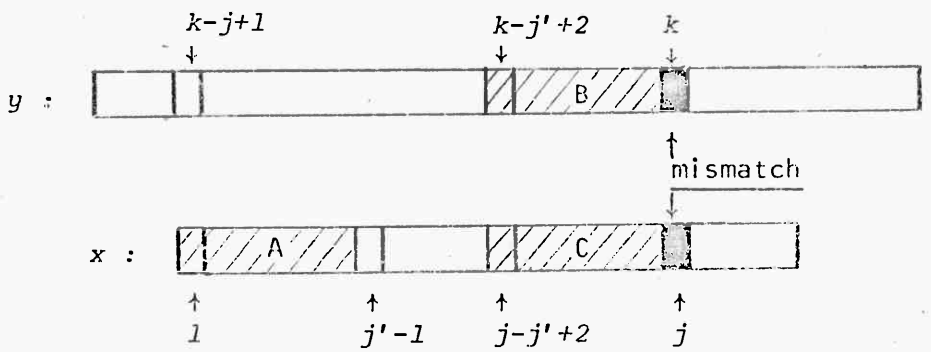
This allows a straightforward *S*-equivalence substitution in the modified procedure to give :-

$$string^{**}(x,y,j,k) \leftarrow item(u,j,x),\ item(v,k,y),\ u \neq v,\ j' \leqslant j,$$
$$\sim(\exists k^*)(k-j+2 \leqslant k^* < k-j'+2,\ prefix(x,suf(y,k^*))),$$
$$prefix(pre(x,j'),\ suf(y,k-j'+2)),$$
$$string^{**}(x,y,j',k+1)$$

Having thus obtained a preliminary procedure which, after a mismatch of x_j and y_k, computes a displacement which, with no loss of solutions, allows x_1 to re-align with $y_{k-j'+2}$ and resumes the comparisons beginning with $x_{j'}$ and y_{k+1}, it is appropriate to consider how the displacement is actually computed in the linear algorithm.

The arrangement proposed by Knuth, Morris and Pratt causes the algorithm to refer to a pre-processed data structure which effectively tabulates, for each possible combination of pointer j and member y_k, the largest value of j' which makes $pre(x,j'-1)$ a prefix of $suf(x,j-j'+2)$ and satisfies the formula $(y_k = x_{j'-1} \leftarrow j'>1)$. This data structure can be constructed from knowledge of x alone by assuming the possible choices of y_k to be either those which are members of x or those which are not members of x; Knuth, Morris and Pratt have shown that the construction is then achievable with an algorithm whose efficiency is linearly dependent upon the length L of x.

The significance of computing j' so as to satisfy the above constraints may become appreciated from inspection of yet another picture, which depicts the state of the algorithm when a mismatch has occurred but the displacement not yet put into effect :-



light-shaded sections A, B and C match at instant when x_j and y_k mismatch

In the diagram of the instant at mismatch, the sections A, B and C are identical. B and C match because of the assumption that *pre(x,j)* has already been matched with that part of *y* with which it aligns. A and C match such as to maximize *j'* (whose least value is *1*). When *x* is displaced after the mismatch, A becomes aligned with B, and $x_{j'-1}$, if it exists, aligns with y_k, which it matches because of the way it is computed in accordance with the Knuth-Morris-Pratt rule given earlier. It follows that, after the displacement, comparisons can resume beginning with $x_{j'}$ and y_{k+1}. The maximization of *j'* and hence of the lengths of A and B determines that no potential solutions are omitted by the displacement of *x* through *j-j'+1* positions, since any intermediate solution would obviously imply a non-maximal value of *j'*.

Some useful consequences follow from the computation of *j'* as just described. Suppose that the predicate *displace(x,y,j,k,j')* holds when *j'* is computed as specified above. A call to a procedure for *displace* can be implemented as a look-up of an assertional data structure which uses the input *x,j* and y_k to determine *j'* ; this data structure can be pre-computed by appropriate bottom-up processing of other procedures which implement the algorithm of Knuth et al. The specification of *displace(x,y,j,k,j')* - requiring *j'* to be the maximum value satisfying *prefix(pre(x,j'-1), suf(x,j-j'+2))* and $(y_k = x_{j'-1} \leftarrow j' > 1)$ - admits simple proofs of the following facts :-

1] $j' \leq j \leftarrow displace(x,y,j,k,j')$

2] $\sim(\exists k^*)(k-j+2 \leq k^* < k-j'+2,\ prefix(x,suf(y,k^*)))$

$\leftarrow displace(x,y,j,k,j'),$
$prefix(pre(x,j),\ suf(y,k-j+1))$

3] $prefix(pre(x,j'),\ suf(y,k-j'+2))$

$\leftarrow displace(x,y,j,k,j'),$
$prefix(pre(x,j),\ suf(y,k-j+1))$

Invoking these by *modus tollens* in response to the calls of the modified procedure then produces the result :-

$string^{**}(x,y,j,k) \leftarrow item(u,j,x),\ item(v,k,y),\ u \neq v,$
$displace(x,y,j,k,j'),$
$prefix(pre(x,j),\ suf(y,k-j+1)),$
$string^{**}(x,y,j',k+1)$

One more step is necessary now to turn this last result into a useful procedure for the linear string searching algorithm. That step has the object of eliminating the call to *prefix* which still remains in that result. The call occurs there of logical necessity, since unless it is satisfied the conclusion *string**(x,y,j,k)* cannot be drawn even though the other calls are satisfied. However, the *context* in which this procedure is invoked is such that the predicate *prefix(pre(x,j),suf(y,k-j+1))* is already satisfied as a result of previous successful matching prior to the mismatch . Thus the call to *prefix* is computationally, if not logically, superfluous. The way to eliminate the call is suggested by our earlier experience in the derivation of the merge-sort program body, where computationally superfluous checks upon orderedness of lists were eliminated by a simple transformation. In the present case, it suffices to specify a new predicate :-

*string***(x,y,j,k) ↔ (string**(x,y,j,k) ← prefix(pre(x,j), suf(y,k-j+1)))*

and use this exactly as in the transformation of the *merge* procedures into *merge** procedures, thereby giving the final program body for the Knuth-Morris-Pratt algorithm apart from the assertions solving calls to *item* and *displace*:-

*string(x,y) ← string***(x,y,1,1)*

*string***(x,y,j,k) ← length(x,w), j>w*

*string***(x,y,j,k) ← item(u,j,x),*
*item(u,k,y), string***(x,y,j+1,k+1)*

*string***(x,y,j,k) ← item(u,j,x), item(v,k,y), u≠v,*
displace(x,y,j,k,j'),
*string***(x,y,j',k+1)*

Execution of these procedures by Prolog-like control gives the linear algorithm. The algorithm is conventionally programmed such that the logic of the two recursive *string**** procedures is encoded within a single 'next move' function which maps *(x,y,j,k)* to *(x,y,j',k+1)*, where *(j,k)* identifies the current comparison and *(j',k+1)* identifies the next. An interesting formulation of the algorithm is given by

Aho and Corasick (1) who treat the 'next move' function as a deterministic finite state automaton which processes the input string y as its input tape. They give proofs of some Algol-like procedures which construct the automaton from x in time linearly dependent upon the length L of x, and then prove that the automaton behaves deterministically. Their treatment is more general than the logic derivation given here in that their algorithm can deal with a set of several keywords simultaneously rather than, as here, just one.

The Sub-linear Algorithm

When *string(x,y)* is solvable, there exists some least $\hat{k}$ for which *string(x,suf(y,$\hat{k}$))* holds. The Knuth-Morris-Pratt linear algorithm finds this solution after exactly $(\hat{k}+L-1)$ comparisons. Here we now consider briefly a remarkable algorithm due to Boyer and Moore (6) which finds this same solution with *fewer* comparisons, for which reason it is called the 'sub-linear algorithm'. It also has the surprising property that, generally, its comparison count *decreases* with increasing length L of x.

The essential idea in the sub-linear algorithm is that of performing the comparisons in the reverse order to that of the naive and linear algorithms. The overall strategy of repeated suffix selection and prefix testing is retained, so that each displacement of x after a mismatch moves x further towards the right-end of y. The particular qualities of the sub-linear algorithm arise partly from the special nature of its method of prefix testing (that is, by the reversed order of comparisons) and partly from the way in which it computes the displacements of x. The logic representation of this algorithm can be developed by appropriately modifying the *string*** procedures :-

```
string(x,y) ← string**(x,y,1,1)

string**(x,y,j,k) ← length(x,w), j>w
string**(x,y,j,k) ← item(u,j,x), item(u,k,y),
                    string**(x,y,j+1,k+1)
string**(x,y,j,k) ← item(u,j,x), item(v,k,y), u≠v,
                    string**(x,y,1,k-j+2)
```

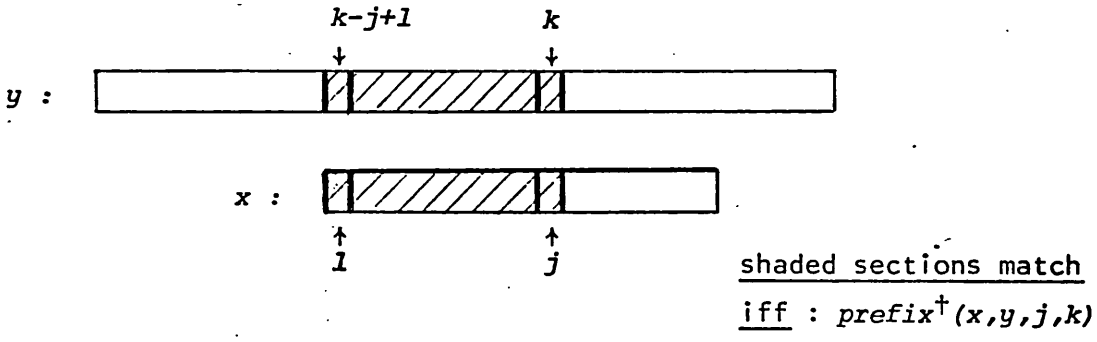
Recall that the purpose of a call *string**(x,y,j,k)* is to show that either $(x_j, x_{j+1}, \ldots)$ matches $(y_k, y_{k+1}, \ldots)$ or else x is a string in *suf(y,k-j+2)*, in accordance with the specifications :-

$$string^{**}(x,y,j,k) \leftrightarrow prefix^{*}(x,y,j,k) \vee string(x,suf(y,k-j+2))$$
$$prefix^{*}(x,y,j,k) \leftrightarrow prefix(suf(x,j), suf(y,k))$$

When it is required to conduct a prefix test in the reverse order, this can be achieved by calling a new procedure $string^{\dagger\dagger}(x,y,j,k)$ specified by :-

$$string^{\dagger\dagger}(x,y,j,k) \leftrightarrow prefix^{\dagger}(x,y,j,k) \vee string(x,suf(y,k-j+2))$$
$$prefix^{\dagger}(x,y,j,k) \leftrightarrow prefix(pre(x,j+1), suf(y,k-j+1))$$

A call $string^{\dagger\dagger}(x,y,j,k)$ has the object of showing that either $(x_j, x_{j-1}, \ldots)$ matches $(y_k, y_{k-1}, \ldots)$ or else x is a string in *suf(y,k-j+2)*. The picture below illustrates the meaning of $prefix^{\dagger}(x,y,j,k)$:-



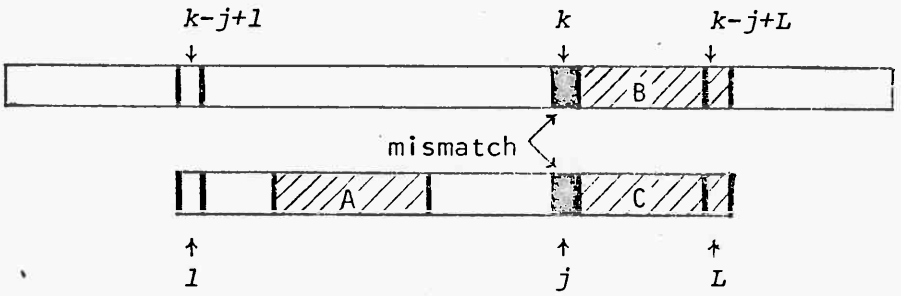
Using the new $string^{\dagger\dagger}$ predicate it is now possible to construct procedures directly analogous to those for *string***. They are :-

$$string(x,y) \leftarrow length(x,w),\ string^{\dagger\dagger}(x,y,w,w)$$

$$string^{\dagger\dagger}(x,y,j,k) \leftarrow j<1$$
$$string^{\dagger\dagger}(x,y,j,k) \leftarrow item(u,j,x),\ item(u,k,y),\ string^{\dagger\dagger}(x,y,j-1,k-1)$$
$$string^{\dagger\dagger}(x,y,j,k) \leftarrow item(u,j,x),\ item(v,k,y),\ u \neq v,\ length(x,w),\ string^{\dagger\dagger}(x,y,w,k-j+w+1)$$

These can all be derived formally without difficulty. When executed to solve *string(x,y)* their overall behaviour is that of the naive algorithm in that they potentially inspect every suffix in y to see if it has x as a prefix. The order of comparisons in each prefix test is inconsequential as far as the asymptotic behaviour of this algorithm is concerned; the worst-case total comparison count continues to exhibit worse-than-linear dependence upon the length L of x.

Top-down execution of the new procedures has the effect that when a mismatch occurs for some pair (x_j, y_k) then the suffix *suf(x,j+1)* will have already been matched with that part of y with which it aligns. This knowledge is exploited in the Boyer-Moore algorithm in such a way that the subsequent displacement of x may be more than one position. Suppose then that execution reaches the state depicted below, in which such a mismatch has just occurred prior to a displacement of x :-



The light-shaded sections B and C match at this point. Moreover, let A be the penultimate occurrence, if any, of C in x. If A exists, then x can be displaced so as to align A with B without omitting any potential solutions. If A does not exist then x can be displaced so as to align x_1 with the member of y which immediately follows the end of B, again with no omission of potential solutions. In each case the comparisons are then resumed beginning with the last member of x. Irrespective of whether or not A exists, there is yet another piece of information which can be used to decide the optimal displacement. Suppose that k'' is the maximum position in x less than j(if any) at which $x_{k''}=y_k$; then x can be displaced so as to align $x_{k''}$ with y_k. If k'' does not exist then x can be displaced so as to align x_1 with y_{k+1}. Again, these displacements omit no potential solutions, and are followed by resumption of comparisons beginning at the end of x.

Boyer and Moore show that x can be pre-processed such that the appropriate displacement (which is chosen to be the largest afforded by the various choices indicated above) can be looked up in a data structure using y_k as the key. In logic this arrangement can be implemented by a procedure call *displace*†*(x,y,j,k,k')*, whose specification allows a proof of the sentence :-

$$(string^{\dagger\dagger}(x,y,w,k\text{-}j+w+1) \leftrightarrow string^{\dagger\dagger}(x,y,w,k'))$$

$$\leftarrow length(x,w),$$
$$prefix(suf(x,j+1),\ suf(y,k+1)),$$
$$item(u,j,x),\ item(v,k,y),\ u \neq v,$$
$$displace^{\dagger}(x,y,j,k,k')$$

The antecedent in the lemma above just implies that $prefix(x,suf(y,k^*))$ is false for all k^* in the range $k\text{-}j+2 \leqslant k^* < k'\text{-}L+1$, proof of which follows easily from the $displace^{\dagger}$ specification. Consequently the antecedent implies that $string^{\dagger\dagger}(x,y,w,k\text{-}j+L+1)$ is completely investigated by displacing x through $k'\text{-}(k\text{-}j+L)$ positions and then investigating $string^{\dagger\dagger}(x,y,w,k')$. As with the linear algorithm, a final transformation is necessary in order to dispense with explicitly testing the condition $prefix(suf(x,j+1),suf(y,k+1))$ after a mismatch, since this will have already been ensured by the way in which procedure invocation is controlled. By specifying a final predicate analogous to $string^{***}$ as follows :-

$$string^{\dagger\dagger\dagger}(x,y,j,k) \leftrightarrow (string^{\dagger\dagger}(x,y,j,k) \leftarrow prefix(suf(x,j+1,\ suf(y,k+1)))$$

the high-level procedures of the sub-linear algorithm are found to be transformed to :-

$$string(x,y) \leftarrow length(x,w),\ string^{\dagger\dagger\dagger}(x,y,w,w)$$

$$string^{\dagger\dagger\dagger}(x,y,j,k) \leftarrow j<1$$

$$string^{\dagger\dagger\dagger}(x,y,j,k) \leftarrow item(u,j,x),\ item(u,k,y),\ string^{\dagger\dagger\dagger}(x,y,j\text{-}1,k\text{-}1)$$

$$string^{\dagger\dagger\dagger}(x,y,j,k) \leftarrow item(u,j,x),\ item(v,k,y),\ u \neq v,\ displace^{\dagger}(x,y,j,k,k'),\ string^{\dagger\dagger\dagger}(x,y,w,k')$$

The two recursive $string^{\dagger\dagger\dagger}$ procedures can be reformulated as a single one employing a slightly more elaborate displacement procedure which behaves as a deterministic 'next move' generator like that used in the Aho-Corasick implementation. It is also worth observing that whereas each call to the recursive $string^{***}$ procedures in the linear algorithm increments the suffix pointer k by 1, thus inspecting every member of y (potentially), this is not true of the recursive $string^{\dagger\dagger\dagger}$ procedures; in general the sub-linear algorithm

inspects fewer than the first $(\hat{k}+L-1)$ members of y, and indeed can also inspect some of them more than once. Its worst-case behaviour is that of the naive algorithm.

CLOSURE

Retrospect

The thesis set out to show that standard FOPL has a substantial and practicable role to play in several important activities associated with logic programming. Its usefulness for specification, derivation and transformation has been especially emphasized and, it is believed, justified by successful application to the various examples presented here. We have used FOPL to specify the relations computed by programs, to formulate useful lemmas about the problem domains of interest and to express the goals of procedure derivations. Throughout these applications, deductive analysis of relations has been treated as the fundamental business of logic program composition as well as of logic program execution. We have assumed that future logic programmers can be trained to a sufficient degree of competence in logical manipulation to allow the expression of such analyses to proceed fluently and naturally. It is our confident expectation that such competence could be instilled quite easily by virtue of the essential simplicity of first order logic. This is not to say that the use of logic in this way will greatly diminish the need for serious intellectual effort in the composition of programs; rather we expect that the programmer who is already capable of formulating intelligent and well-organized ideas about his intended algorithms will, after suitable training, find logic to be a more satisfactory means of expressing those ideas than conventional computational languages.

Little attention has been given in the present research to the prospect of automating syntheses and verifications of logic programs. This is not because such a possibility was considered to be either unimportant or ultimately unattainable, but rather because it seemed more urgent to establish that logic is practicable as a human-oriented programming language. Unless its credibility can be proven in this respect first, it will attract little immediate attention from the existing programming community and so diminish the probability that

researchers will become motivated to devise useful mechanized logic programming aids. Apart from this consideration, it has to be recognized that insofar as many useful logic programs may require the inclusion of arbitrarily 'deep' theorems as procedures, the *general* problem of fully automating logic procedure derivation approximates to that of automating the derivation of much of mathematics; we cannot realistically regard this problem as capable of short-term solution given our existing state of knowledge. In view of these considerations it appeared a more sensible objective to consolidate a comparatively informal and empirical corpus of experience in program derivation without demanding rigid adherence to any particular set of derivation rules or strategies. Nevertheless it is already clear that there is much scope for developing useful mechanized aids for such tasks as checking given derivations or interacting with the programmer's decisions whilst he chooses amongst alternative paths through the derivation search space. Existing aids for deriving programs in other formalisms, such as those surveyed presently, will doubtless contribute useful strategies for these purposes.

Related Research

Although the earlier parts of the thesis have included quite a number of citations of related work, it is useful at this concluding stage to review in a little more detail those projects undertaken by other researchers which afford reasonably close comparison with our own. The general field of program synthesis is naturally very wide, but discussion is confined here to projects whose object is to provide for deductive derivations of programs from complete specifications of their computed relations. Therefore we omit comparisons with methods such as Kodratoff's (45) in which specifications are incomplete or in which the derivation process depends upon highly specialized mathematical analyses bearing little relation to the conventional approach to program synthesis.

The early contributions of theorem proving to program synthesis have already been surveyed briefly in Chapter 1. The work of Green on answer-extraction from resolution proofs formed the basis upon which both he and Waldinger subsequently implemented various synthesizers capable of generating very simple assignment programs from the bindings induced by proofs of their input-output relations. Later on an

interesting advance was made by Manna and Waldinger in their discovery of the utility of induction axioms over the data domains of interest for enabling the construction of recursive and iterative programs. At that time they were not particularly optimistic about the prospects for autonomous synthesizers. They noted the probable need for restrictive strategies in order to control the manipulation of the specification axioms, but also recognized the difficulty of designing these so as to be sufficiently general to cope with a variety of semantic domains. They were aware, too, of the potential usefulness of future interactive synthesizers.

Since these early beginnings, Manna and Waldinger have actively pursued their synthesis work at Stanford. Their 1975 paper (61) indicates several changes in approach since the 1971 report (60) and additionally describes a partially completed implementation. Rather than using FOPL to express program specifications, they choose a high-level quasi-procedural language in which to describe the input problems to the synthesizer; the language is not defined formally and is viewed as arbitrarily extendible by the additions of new general constructs and domain-specific notations. This arrangement represents a significant departure from their previous use (60) of purely descriptive, tightly-formalized axiomatic specifications, although the new specification language does admit a certain amount of logical symbolism such as quantification and basic connectives. Consequently a considerable sacrifice of uniformity and simplicity in both syntax and semantics is incurred in their abandonment of FOPL, although it might be claimed that their input problem specifications usefully suggest initial abstract algorithms awaiting suitable refinements; however, we would argue that similar effects could be obtained just as convincingly in FOPL through the use of appropriate logical styles. The language chosen for the target programs generated by the synthesizer is intended to be essentially LISP-like but capable also of supporting side-effect mechanisms like destructive assignment. The generation of the target programs proceeds incrementally by application of various transformation rules which refine the problem description through successive stages towards an executable program. The transformations are implemented by procedures written in some language suitable for encoding reasoning tactics such as QLISP; these are summoned by pattern-directed procedure invocation induced by sub-expressions of the current problem description, such that the

overall effect is to gradually replace descriptive expressions by suitable algorithmic constructions. No discussion is given by Manna and Waldinger in this paper of what strategic principles are assumed to govern the choice of transformations when several are simultaneously applicable; intelligent choice in such cases is obviously crucial to both the efficiency of the synthesis and the usefulness of the output program.

The transformation rules used by Manna and Waldinger are simple and few in number. They provide chiefly for the construction of conditionals and recursions, for solution of conjoint goals and for introducing side-effects. Their mechanism for recursion construction is especially interesting and appears, in one guise or another, in various other researchers' synthesis systems. It is invoked upon recognition that the current description of the problem's goal contains as a sub-expression some substitution instance of the goal's definiens, thus allowing that instance to be replaced by an appropriately instantiated call to that goal; this clearly results in a recursive description of the goal. Manna and Waldinger also employ a check to ensure that a computation induced by the recursion will terminate by appealing to some well-ordering defined upon its argument domain; their transformation process therefore preserves total - rather than just partial - correctness.

Another interesting technique employed in their system is that of generalization of specifications. Usually this is summoned when it appears impossible to construct certain recursions using just the goal predicates or functions initially given; generalization entails the reformulation of the goal's description, typically by introducing new parameters into it, in such a way as to permit construction of the desired recursion. Instances of this have already been seen in this thesis. For example, the use of the *reverse* predicate in Section 6.1 precludes a recursive description of the list reversal problem in which calls to *append* are absent; but use of the *reverse** predicate, which generalizes *reverse* through the device of an additional parameter, does allow such a description. This technique often results in improvements to efficiency such as the replacement of recursions by iterations. Manna and Waldinger offer some loose guidelines for recognizing opportunities for making simple generalizations, but they do not yet possess any precise characterization of the technique which

would allow either its automatic application or its evaluation in terms of computational advantage. In fact the particular kinds of reformulations which they examine are just aspects of a much more general problem which, in our formalism, is manifested as that of choosing the 'right' predicates for problem specification. For example, the elaborations of the *string* predicate used in our refinements of the text-searching problem in Chapter 7 entail the introduction of new argument structures apparently beyond the scope of Manna's and Waldinger's simple generalization technique and yet are motivated by comparable intentions, namely the construction of sophisticated recursions whose logic induces special behavioural effects. Their paper likewise examines the list reversal problem and also shows how the method can be successfully applied to the rather harder problem of generalizing a pattern-matching algorithm into an algorithm capable of computing most-general unifiers. We concur with their view of the importance of gaining an understanding of the underlying principles of appropriate predicate formulation in order to allow more intelligent syntheses and to clarify the pragmatic distinctions between alternative programming styles.

A more recent working implementation of a synthesis system called 'DEDALUS', incorporating the ideas previously described, is reported by Manna and Waldinger in a 1977 Stanford Report *[Report No. STAN-CS-77-630 : "Synthesis : Dreams => Programs"]*, and a further implementation called 'SYNSYS' is reported in their IJCAI-77 paper (62). The specification language used there continues to be a somewhat arbitrary mixture of logical, mathematical and algorithmic notations. The target language is pure LISP (having no side-effect features) and QLISP is used to encode more than a hundred transformation rules. They regard the system as being essentially 'deductive', although the relationships which prevail between their specifications and target programs are not those of logical implication as in our own derivation methodology. Nevertheless it seems clear that a suitable axiomatic formulation of their rules and specifications could be devised so as to enable their syntheses to appear as deductions employing logical equivalence substitution as the principal refinement mechanism.

Manna and Waldinger emphasize that they do not have a strong prior commitment to any particular specification language or programming language. They regard their project as belonging more to research in

artificial intelligence than to research in general programming methodology, and their interests are primarily in elucidating reasoning strategies rather than in advocacy of new programming formalisms. However, this is not to say that their approach to program synthesis is neutral with respect to the choice of languages, since there are several respects in which its development has been affected by the particular choices which they have in fact made. Firstly, because of their use of different languages for expressing programs and specifications, the transformation rules at the heart of their system rely for their justification upon establishing relationships between two distinct semantics. Although Manna and Waldinger would like to regard their syntheses as 'deductive', the deductive relationships actually exploited do not appear explicitly in the successive object-level problem representations developed by the synthesizer - instead they are only implicit in the 'crossing' axioms which have presumably been invoked in order to justify the procedures which implement the various transformations. This arrangement tends to make the logical basis of the transformations less visible (and so less obvious) than we should ideally desire. Secondly, they recognize that program modification represents an important aspect of synthesis, so that a comprehensive synthesizer should be capable of accepting as *input* information encoded in the target programming language. Insofar as their system already accepts somewhat procedural specifications it would seem likely that it could also accept LISP-like programs as input and then proceed to modify them, and in fact they do describe some hypothetical examples of this. However, such an arrangement requires that the semantics of the specification language should fully incorporate that of the programming language, and it could be argued that the use of a conventional language having semantically awkward features like destructive assignment (unlike pure LISP) would make the complete formalization of the synthesis system unduly cumbersome. Thirdly, their choice of formalism constrains their syntheses to be input-output deterministic. Their specifications are functional rather than relational and their target programs are function evaluators. This means that distinct syntheses are necessary in order to procure programs which investigate various input-output arrangements of some given relation; by contrast, we often find that a derived logic procedure set is suitable for solving different input-output permutations of the goal arguments.

The various papers by Manna and Waldinger suggest some ambivalence on their part towards adopting a single formalism such as logic. The 1971 report (60) uses FOPL for specification but not in the way that we do, since they employ it primarily to construct theorems expressing the *existence* of solutions satisfying the partial correctness formula rather than for simply defining the relations of interest. Their more recent specification languages are claimed to include FOPL, yet still they clearly do not intend to deploy it in the manner used for logic program specification. The 1971 report anticipates the possible adoption of partial function logic instead, presumably for both specification and target languages, but this idea does not seem to have been pursued further. *[*Interestingly, the contemporaneous work of Burstall and Darlington, considered presently, does employ the language of recursive functions as a single uniform formalism for program derivation.*]* In the 1977 Stanford Report cited earlier, Manna and Waldinger refer to papers by Kowalski and by Clark and Sickel, but mistakenly assert there that logic specifications are restricted to clausal form. They question whether logic (clausal or otherwise) is sufficiently expressive to serve as a specification language in view of its preclusion of the algorithmic constructs used in their own problem formulations; but this doubt probably arises because of their different conception of logic's role in specification, as they apparently wish to make their specifications somewhat algorithmic in character.

In summary, the work of Manna and Waldinger has usefully influenced the approach taken here to logic program derivation in that it has helped to clarify the tactics necessary for introducing recursions and has confirmed the importance of the generalization technique. Devices such as side-effects are, of course, regarded as implementation features in our treatment rather than as matters deserving representation in the logical development of programs. Their loosely-defined specification language appears to be unnecessarily extensive and results in a loss of semantical simplicity and uniformity, and we would hope that they will eventually reconsider the use of logic as the principal formalism.

In many ways the work of Manna and Waldinger since the early 1970's has been paralleled by that of Burstall and Darlington, whose studies of program derivation are described in various papers (9, 10, 19). These studies have their origins in Darlington's earlier doctoral research into systematic program improvement, and have now culminated in

a nicely organized and semantically clear implementation of a system capable of transforming programs expressed as sets of recursion equations. The notion of syntheses beginning with purely descriptive *specifications* is not prominent in Darlington's approach, which is instead essentially directed to the transformation of naive *programs* into more sophisticated ones, using a uniform notation throughout. The input equations do, of course, possess a perfectly straightforward declarative semantics, being no more than function definitions, yet are intrinsically more procedural in character than typical logic specifications; for instance, in order to define some function by reference to all members of some set, Darlington's formalism must resort to a definition which has the appearance of a procedure which recursively inspects that set's individual members, whereas in standard FOPL we can use a universal quantifier instead and thereby avoid a recursive specification.

Darlington's current implementation accepts an initial set of function definitions as input, together with any lemmas which the user considers might be useful to the ensuing transformations. The latter arise through a succession of rule applications including (i) symbolic execution ('unfolding') of the given definitions (treating them collectively as a function evaluation program using call-by-name invocation) for particular argument instances selected by the user, (ii) rewriting definitions using given lemmas and certain built-in laws and (iii) replacement of definiens sub-expressions by calls to the given functions ('folding'), which is tactically similar to the recursion-introduction method of Manna and Waldinger. Some simple, general and non-deterministic heuristic algorithms are proposed by Burstall and Darlington (10) for scheduling these various phases of the transformation process and are shown to be effective for a variety of problems such as list reversal, generation of factorial tables and comparison of tree frontiers.

Some reviews by other researchers have mistakenly suggested that the implementation described in the 1977 paper (10) is interactive and necessarily a program improver. In reality the user has to contribute much of the intelligence required for procuring the desired output by selecting appropriate definitions, lemmas and instantiations as initial input, whereupon the transformations are executed autonomously and exhaustively without further user intervention. [However, it may be

the case that Darlington - who continues to implement enhancements - has since introduced some truly interative capability.] Furthermore, as Burstall and Darlington admit, there is no precise reason to expect that - in general - their transformations will result in more efficient programs, although there are good intuitive reasons for supposing that certain steps (like 'folding') are more likely to eliminate computational redundancies than to introduce them.

Burstall and Darlington also recognize the important role of generalization and other kinds of redefinition although, like Manna and Waldinger, they have not yet been able to fully characterize or automate such capabilities. Altogether the contemporaneous but independent approaches taken by these two research groups seem to have similar scope and power, and have identified much the same general principles underlying the process of transformation by symbolic execution. However, Darlington's implementation benefits very substantially from his use of a formalism having a simple and precise semantics which allows exceedingly transparent application of the transformation rules. Moreover, because his function definition language can be trivially paraphrased in Horn clause logic, his implementation could easily and usefully be adapted as an automated aid to logic program derivation, a possibility foreseen some years ago by Keith Clark. A particularly interesting feature incorporated by Darlington is a built-in ability of the matching routine used for 'folding' and 'unfolding' to exploit special user-asserted function properties like associativity, thereby providing for more sophisticated kinds of pattern-directed invocation than that used in this thesis for making subformula substitutions - our treatment has explicitly encoded such properties as derived lemmas which are summoned just as though they were arbitrary specification axioms without conferring any enhancements upon the normal matching process.

The question of whether Darlington's approach could also cater for specifications presented in standard FOPL - with the object of synthesis 'proper' - is a more difficult one. We know that 'folding' and 'unfolding' are just instances of what has been termed herein as goal substitution, whilst our goal simplifications are like some of Darlington's rewriting laws. However, some of our transformations which are performed during logic procedure derivation - such as

conditional equivalence substitution - seem to have no counterpart in his system, even though they are undoubtedly powerful and frequently-used devices for invoking facts from the specification set. Further studies will have to be pursued before it will be possible to assess the applicability of Darlington's implementation to the general problem of logic program derivation using standard FOPL.

Darlington's transformation system does not try to measure or compare the computational efficiencies of the programs which it generates. By contrast, Wegbreit (86) has attempted to integrate an inference system like Darlington's with provisions for analysing programs (albeit somewhat superficially) in order to identify sources of inefficiency. With these arrangements, an initial input program is analysed so as to identify those segments of its text which are responsible for redundant computational effort; these segments are then regarded as targets for simplification using rules like 'folding', 'unfolding', generalization and such-like with the object of improving the program's overall run-time performance. On the whole, the kind of improvement which can be obtained by Wegbreit's system only requires rather modest transformations to the program text and does not involve a radical alteration of the overall algorithm structure; for example, Wegbreit sees no way of using it to transform a bubble-sort program into a quick-sort program. Input programs are presented in a LISP-like notation which is not greatly removed from Horn clause logic, and so it is possible to envisage the application of this simple improving system to logic programs.

A synthesis system of considerable power, called 'PECOS', has been implemented at Yale University; a brief outline of its capabilities is given by Barstow (2). 'PECOS' accepts some very abstract algorithm as an input specification and then generates a tree of 'refinement' sequences terminating in INTERLISP programs which implement the algorithm concretely; the tree is grown during the synthesis by summoning applicable rules from a catalogue of more than four hundred, this being an extremely heterogeneous mixture of refinement rules encoding knowledge about the logical and implementational properties of various classes of algorithms and data structures. Operation of the system can be controlled interactively to allow the user to decide which derivations in the developing tree are to be continued or abandoned. Because the built-in rules are well-informed about the

properties of sets, lists, arrays, permutations and orderings, 'PECOS' has been able to synthesize a wide range of very concrete sorting algorithms. The versatility of 'PECOS' seems to be chiefly attributable to its large catalogue of specific rules rather than to the use of powerful general strategies.

An approach to program synthesis which is rather different from those previously considered has been taken by Bibel (3) in Munich. He uses standard FOPL as a specification language but not quite in the way that we do; instead he uses it to construct input-output specifications like those used by Manna and Waldinger, having the general form :-

$$(\forall\tilde{x}\exists\tilde{y})\,(\textit{output-predicate}(\tilde{y}) \leftarrow \textit{input-predicate}(\tilde{x}))$$

and thereby fixes the input-output status of the variables. The matrix of a formula like that above is then interpreted by Bibel as a definiens for a *function* which maps $\tilde{x}$ to $\tilde{y}$. Such specifications are intended to be purely descriptive and thus undisposed towards any particular algorithms for computing their associated functions. Bibel is not especially concerned with the choice of target language and resorts to a loosely-defined ALGOL-like notation for the expression of his derived programs. Synthesis proceeds by summoning a rather curious collection of transformation rules, of which some are activated in response to highly specific syntactical structures in the definiens being transformed, whilst others are applicable only over certain semantic domains such as set theory; it is expected (by Bibel) that many more rules will have to be devised in order to provide a reasonably comprehensive synthesis tool. Bibel attaches much significance to the precise arrangement of quantification over the definiens' variables, associating distinct rules with distinct groupings of quantifiers. Amongst his rules one can discern provisions analogous to Darlington's 'folding' and rewriting tactics, but it is clear that he does not view his transformations as symbolic executions. Some small account is given to matters of efficiency by, for instance, comparing the cardinalities of alternative sets to be searched, but these arrangements appear to be rather idiosyncratic. Bibel also believes that certain *sequences* of rule application are sufficiently powerful to synthesize quite broad classes of algorithms, and so his system tends to be rather more deterministic than those of Waldinger and Darlington. Altogether it is rather difficult to reach

a clear appraisal of the relationship which Bibel's work bears to that of the latter researchers or ourselves, since he provides no such comparison himself and, in his reports to date, leaves much technical and motivational detail unspecified. No implementation has been pursued yet, because Bibel has not been able to gain access to computer facilities at his own institute.

The work of Keith Clark is undoubtedly closer to the research described in this thesis than any of the other projects for program derivation previously discussed. Comments upon both his approach and his examples appear in earlier chapters and so it is not necessary to recount them at length here. Suffice it to recall that he also uses standard FOPL for specifying relations and derives logic programs from them using inference rules like those of Darlington. Clark's initial ideas on program derivation appear in a draft paper for IJCAI-77 (15) in which he chooses the *subset* problem to illustrate logic procedure derivation; there he also sketches the relationship between the latter and the inductive verification method which he had previously developed with Tarnlund. In this draft paper he presents his derivation method as one specifically applicable to problems dealing with inductively definable sets, but his later papers drop this undeserved emphasis. The final IJCAI-77 paper uses a slightly more interesting example of a program which assigns a *yes/no* answer to an output variable in response to the question of whether a given element belongs to a given list. This example also appears in a rather more lengthy report (12). In that report Clark presents an *append* derivation and derives a number of iterative programs for the *fact*, *reverse* and *fib* relations. The *append* specification which he uses and describes as "intuitively correct" is, in fact, erroneous for the same reason as is his *perm* specification in (13) in that, as explained in Chapter 7, it fails to properly conserve the multiplicities of list members. [However, Clark and Darlington have very recently revised the paper (13) to give a new version entitled *"Algorithm Classification Through Synthesis"* in which they correct their *perm* specification.] It is something of an irony that proponents of logic programming, including the present writer, occasionally present (by accident) confident derivations based upon "intuitively correct" but nonetheless erroneous specifications; this is a cautionary reminder that the potential clarity of logic does not in itself render us immune from the import of Russell's (75) dictum : *"obviousness is the enemy of correctness"*.

Clark has consistently emphasized his treatment of program derivation as symbolic execution, although his presentations of examples do not usually expose this theme very clearly and often appear somewhat disorganized. This impression is given mainly by his seemingly unsystematic treatment of definiens manipulation. Admittedly, it is often very difficult to cast the latter convincingly into the format of a program execution when using standard FOPL, as is reflected by the somewhat non-uniform nature of the inference rules used in this thesis. Because FOPL is inherently less uniform than the notation of recursion equations, Clark's often-expressed (but so far unrealized) intention to implement a practical symbolic executor analogous to Darlington's will prove very difficult to fulfil until we possess a strong problem-solving interpretation to guide and justify all the various syntactical manipulations which FOPL seems to demand. Another significant problem confronting implementation of logic program synthesizers is that of dealing properly with lemma generation. It seems very difficult to organize this as a top-down activity which can be assimilated naturally into the main-stream of the derivation process. Clark also recognizes that the choice of lemmas and the scheduling of their invocations *"is where some of the cleverness comes into the program syntheses"*; this is, if anything, an understatement. Perhaps one day we shall have inference systems which dispose of the need to conduct preliminary derivations of lemmas; the role of lemmas, with the question of whether they are really necessary at all, is an appropriate topic for future research.

Neil Murray at Syracuse University has recently reported (63) an interesting proof procedure for a quantifier-free subclass of FOPL whose inference system he calls "NC-resolution", that is, non-clausal resolution. This is clearly capable of mechanization although the choice of practical control strategies for it remains undetermined. Murray shows how the inference system (which he proves complete) can be used to derive Kowalski's *fact** program from specifications of the *fact*, *times* and *fact** relations, once these have been rewritten in his chosen NC-syntax. *[*Actually his initial *fact* specification is erroneous due to the omission of an existential quantifier from the definiens; fortuitously this does not affect the correctness of the particular derivations which he pursues.*]* Murray's work provides a nicely-judged intermediary between adaptations of Darlington's system for Horn clause logic and inference systems for the complete

standard formulation of FOPL, and it will be interesting to find out how well it deals with more difficult examples.

Topics for Future Research

The present project must be seen as only a preliminary exploration of logic program derivation, although it has already yielded much useful experience and clarification. Many important questions remain open to future study. Primarily, we need to investigate more thoroughly the question of which kinds of inference consistently prove to be the most intuitive and practicable for problems expressed in FOPL. Whilst there seems to be little doubt that such inferences must entail various kinds of substitutions for definiens subformulas, the most desirable preconditions for these substitutions are at present unknown. In most of the examples studied so far, which use fairly simple rules like conditional and unconditional equivalence substitution, it has appeared necessary to engage firstly in a certain amount of lemma generation, often with some degree of sacrifice in goal-directedness. It is possible that a more thorough study of the fundamental logical structure of such examples will suggest alternative rules or ways of writing specifications which will allow more clearly motivated derivations. This, then, is certainly a matter which we shall pursue in the short term : improvements to the inference system.

We further need a better understanding of logic programs themselves in order to discriminate intelligently between alternative targets for derivation. Even when the control component is fixed, there may exist a number of substantially different logic components which, with that control, all give essentially the same algorithm. Some of these logic components may be much more difficult to derive than others or may require more subtle specifications. It would be useful to discover general characteristics of procedure sets which are both practicable to derive and efficient to execute. Furthermore, it is to be expected that advances in implementation technology will lead to interpreters supporting richer control mechanisms than those found in the Prolog family. The ability to vary control components adds an extra level of complexity to the investigation of alternative programming styles. Throughout the thesis it has been assumed that control is Prolog-like, because this reflects the current position regarding practical program implementation. This must have biased the

investigation to some extent in that many of the derivations were deliberately steered towards procedures giving useful sequential computations. Significantly different assumptions about control might have led to different impressions about suitable inference systems for procedure derivation.

At present it is not easy to foresee which control strategies will eventually predominate in automatic computation, whether the latter be instigated by logic programs or otherwise; future developments here depend partly upon which kinds of new processors become available. It would seem unduly presumptuous to believe, as Luckham and others (56) seem to do, that efficient execution of logic programs will require the existing apparatus of conventional control constructs, especially at a time when the adequacy of these is already being questioned in relation to conventional programs and processors. One topic which clearly deserves investigation is the nature and derivation of logic programs intended for concurrent, rather than sequential, processing. At present there is growing interest in the problem of verifying concurrent, but otherwise conventional, programs; we may hope to confirm in due course that their logic program counterparts will prove easier to verify, just as is the case with sequential programs. The advantages of separating logic from control can be expected to withstand quite radical developments in execution strategies, but this conjecture must be tested against experience.

Although logic programming has been presented in this thesis as essentially to do with deductively analysing computed relations, it is clear that this activity must be guided by intelligent consideration of run-time behaviour. Superficially it might appear that FOPL does not provide for explicit representation of that part of the programmer's reasoning which deals with assessing the efficiency of programs, as though such reasoning were necessarily extralogical. From this viewpoint it might then seem that FOPL was less useful for expressing computational knowledge than the various 'algorithmic' logics which have been recently developed with the express intention of formalizing inferences about *computations*. An interesting topic for research, then, is the comparison of these new formalisms with FOPL in order to find out whether the former really can provide the programmer with better program-reasoning tools than the latter. Kowalski has recently studied the problem of treating FOPL as both

object language and metalanguage, using an interfacing predicate which explicitly refers to axiom sets, conjectures, proofs and control strategies. This interface appears to have considerable power for expressing metalogical problems. In particular we can see that it might bring deductive reasoning about computations within the ambit of FOPL, thus countering the proposition that such reasoning demands new systems of logic. Kowalski's research on this topic might therefore form the basis for FOPL inference systems which take account of computational efficiency as well as ensuring the correctness of programs, thereby opening the prospect of a single comprehensive tool for reasoning about all aspects of programs.

We shall continue to use program derivation for the purpose of elucidating the logical relationships between members of algorithm families. The family of sorting algorithms examined in this thesis really encompasses only minor variants of merge-sort. By contrast, we already know from Darlington's earlier studies (20) that certain other sorting algorithms like bubble-sort involve some subtle difficulties not encountered in the merge-sort sub-family, and so these certainly deserve future study. The text-searching algorithms presented here also need much more scrutiny in order to clarify the role of the various lemmas which appeared to be crucial to their respective derivations. It is known that Bibel has been recently examining this family using his derivation system, and so it will be interesting to compare his treatment with our own. Also in Munich, Lothar Schmitz (77) has recently synthesized a family of difficult transitive closure algorithms (though not using logic) which we may investigate in due course.

GLOSSARY

The meanings of the principal predicates used in the thesis are expressed informally below.

append(x,y,z)	appending list y to list x gives z
append(u,y,z)*	appending list y to list (u) gives z
cardin(x,w)	set x has cardinality w
consec(u,v,y)	element v is consecutive to u in list y
count(y,w)	list y has w distinct members
delete(u,x,y)	deleting element u from list x leaves y
duplic(x)	list x contains duplicates
duplic(x,y)*	either list y contains duplicates or lists x and y share a common member
elem(u,i,j,x)	u is the matrix element x_{ij}
embed(x,y,z)	palindrome x is symmetrically embedded in palindrome y to give z
empty(x)	x is the empty set
empty-list(y)	y is the empty list
enter(u,v,x,y)	y is the set union of $\{(u,v)\}$ with x
entry(u,v,x)	(u,v) is a member of set x
equal(x,y)	lists x and x are equal
equiv(x,y)	sets x and y are equivalent
fact(u,v)	v is the factorial of u
fib(u,w)	u is the w^{th} Fibonacci number
filter(x,y)	deleting all duplicates from list x leaves y
first(x,u)	list x has first member u
go(x)	node x (in some graph) is reachable
go(x,y)*	node y is reachable if node x is
insert(u,x,y)	ordered-insertion of u in ordered-list x gives y
item(u,i,x)	u is the i^{th} member of list x
kount(y,w)	list y has w members
label(u,i,x)	u is the ith label in the frontier of tree x
last(x,u)	list x has last member u
length(x,z)	list x has length z
lowerbound(u,x)	u is a lower bound for set x

Contd...

merge(x,y,z)	merging ordered lists *x* and *y* gives *z*
middle(x,y)	list *y* is the middle of list *x*
min(u,x)	*u* is the minimum member of set *x*
occurs(u,w,y)	element *u* has *w* occurrences in list *y*
ord(y)	list *y* is ordered
palin(x)	list *x* is a palindrome
palin(x,z)*	appending list *x* to the reverse of list *z* gives a palindrome
partition(x,y,z)	set *z* is partitioned into sets *x* and *y*
partition(u,y,z)*	set *z* is partitioned into sets {*u*} and *y*
perm(x,y)	list *y* is a permutation of set *x*
pick(u,v,z)	*u* and *v* are members of set *z* satisfying $u<v$
plus(x,y,z)	*z* is the sum of numbers *x* and *y*
prec(u,v,x)	member *u* precedes *v* in list *x*
prefix(x,y)	string *x* is a prefix of string *y*
reverse(x,y)	list *y* is the reverse of list *x*
reverse(z,x,y)*	appending list *x* to the reverse of list *z* gives *y*
same-frontier(x,y)	trees *x* and *y* have identical frontiers
select(u,x,y)	set *y* is the set union of *x* with $\{u \mid min(u,y)\}$
singleton(x,u)	set *x* is the singleton {*u*}
size(x,i,j)	matrix *x* has *i* rows and *j* columns
smaller(x,y)	all members of set *x* are < all members of set *y*
spans(u,v,x,y)	*u* and *v* are respectively members of sets *x* and *y*
string(x,y)	string *x* is a substring of string *y*
subset(x,y)	set *x* is a subset of set *y*
table(x,z)	*x* is the set {*(0,0!), ..., (z,z!)*}
table(x,w,z)*	*x* is the set {*(w,w!), ..., (z,z!)*}
*table**(x,w,v,z)*	*x* is the set {*(w,v) , ..., (z,z!)*} <u>if</u> *w! = v*
times(x,y,z)	*z* is the product of numbers *x* and *y*
unit-list(x,u)	*x* is the unit list *(u)*
union(x,y,z)	*z* is the set union of sets *x* and *y*
union(u,y,z)*	*z* is the set union of sets {*u*} and *y*
$u \in x$	*u* is a member of set *x*
$u \hat{\in} y$	*u* is a member of list *y*

BIBLIOGRAPHIC NOTE

A brief guide is given here to the literature of logic programming relevant to the thesis. The new reader of this literature should firstly consult Kowalski's publications (49,50) to learn how problems may be solved using resolution and to gain appreciation of the scope and expectations of logic programming. This may then be consolidated by reference to van Emden's paper (23). Probably the best description of the Prolog system in English is that given in Warren's reports (82,83,84,85), based very much upon the implementation at Edinburgh University. Practical experience with Prolog is described by Bundy (7,8). A readable account of logic program execution using connection-graph systems instead is presented in an early paper by Tarnlund (79). More recent papers on logic programming generally are those by Clark and Kowalski (14,51). This covers the principal literature of the general field.

Papers concerned specifically with logic program derivation are those by Clark (12,13,15) and by Hogger (38,39,40). The paper by Clark and Tarnlund (16) is an important contribution to logic program verification by other means than derivation. Systems for program derivation in other formalisms are described by Manna and Waldinger (60,61,62), by Burstall and Darlington (10) and by Bibel (3).

Helpful textbooks in computational logic are those by Nilsson (67) and (especially) by Chang and Lee (11). Kowalski is currently preparing a book which revises the original report (49) and which will doubtless become a standard text on logic for problem solving. Wolfgang Bibel is currently writing a book on the applications of logic, and Manna and Waldinger are also preparing a book on program synthesis.

REFERENCES

1] *Aho, A. V. and Corasick, M. J.* Efficient string matching: an aid to bibliographic search. Comm. ACM, 18 No. 6, 1975.

2] *Barstow, D. R.* Experience with a refinement paradigm in a knowledge-based automatic programming system. Proc. AISB/GI Conf. on Artificial Intelligence, Hamburg, July 18-20, 1978.

3] *Bibel, W.* On strategies for the synthesis of algorithms. Proc. AISB/GI Conf. on Artificial Intelligence, Hamburg, July 18-20, 1978.

4] *Black, F.* A deductive question-answering system. Doctoral Dissertation, Harvard, June 1964.

5] *Bledsoe, W. W.* Non-resolution theorem proving. Research Report ATP-29, Automatic Theorem Proving Project, Univ. of Texas at Austin, 1975.

6] *Boyer, R. S. and Moore, J. S.* A fast string searching algorithm. Comm. ACM, 20 No. 10, 1977.

7] *Bundy, A.* My experiences with Prolog. DAI Working Paper No. 12, University of Edinburgh, 1976.

8] *Bundy, A. and Welham, R. K.* Utility procedures in Prolog. DAI Occasional Paper No. 9, University of Edinburgh, 1977.

9] *Burstall, R. M. and Darlington, J.* Some transformations for developing recursive programs. Proc. Int. Conf. on Reliable Software, Los Angeles, California, pp 465-472, 1975.

10] *Burstall, R. M. and Darlington, J.* A transformation system for developing recursive programs. J. ACM 24(1), 1977.

11] *Chang, C-L. and Lee, R. C-T.* Symbolic logic and mechanical theorem proving. Academic Press, 1973.

12] *Clark, K. L.* The synthesis and verification of logic programs. Research Report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London, 1977.

13] Clark, K. L. and Darlington, J. Algorithm analysis through synthesis. Research Report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London, 1977.

14] Clark, K. L. and Kowalski, R. Predicate logic as programming language. Research Report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London, 1977.

15] Clark, K. L. and Sickel, S. Predicate logic : a calculus for the formal derivation of programs. Proc. IJCAI-77, 1977.

16] Clark, K. L. and Tarnlund, S-A. A first order theory of data and programs. Proc. IFIP Congress, 1977.

17] Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P. Un systeme de communication homme-machine en francais. Rapport Preliminaire, Groupe de Researche en Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, 1972.

18] Dahl, O-J., Dijkstra, E. W. and Hoare, C. A. R. Structured Programming. Academic Press, 1972.

19] Darlington, J. Application of program transformation to program synthesis. Proc. Symp. on Proving and Improving Programs, Arc-et-Senans, France, pp. 133-144, 1975.

20] Darlington, J. A synthesis of several sorting algorithms. DAI Research Report No. 23, University of Edinburgh, 1976.

21] Davis, M. and Putnam, H. A computing procedure for quantification theory. J. ACM 7 , 1960.

22] Dijkstra, E. W. Correctness concerns and, among other things, why they are resented. Proc. Int. Conf. on Reliable Software, Los Angeles, California, pp. 546-550, 1975.

23] van Emden, M. Programming with resolution logic. Machine Intelligence 8 , 1977.

24] van Emden, M. and Kowalski, R. The semantics of predicate calculus as a programming language. J. ACM, 23 (4), 1976.

25] Floyd, R. W. Algorithm 245, TREESORT 3. Comm. ACM 7 No. 12, 1964.

26] Floyd, R. W. Assigning meanings to programs. Proc. Symposia in Applied Mathematics, Vol 19, Amer. Math. Soc., pp. 19-32, 1967.

27] *Gilmore, P. C.* A proof method for quantification theory. IBM J. Research and Development , 4 , pp. 28-35, 1960.

28] *Green, C. and Raphael, B.* The use of theorem-proving techniques in question-answering systems. Proc. ACM 23rd Nat. Conf., pp. 169-181, Brandon Systems Press, Princeton, N. J. , 1968.

29] *Green, C.* The application of theorem proving to question answering systems. PhD Thesis, Stanford University at Stanford, California, 1969.

30] *Green, C.* Application of theorem proving to problem solving. Proc. IJCAI Conf., Washington D.C., 1969.

31] *Green, C.* Progress report on program-understanding systems. Stanford Memo AIM-240, Computer Science Dept., Stanford University, 1974.

32] *Green, C. and Barstow, D.* Program synthesis for efficient sorting. A.I. Lab. Report, Computer Science Dept., Stanford University, 1977.

33] *Hayes, P. J.* Computation and deduction. Proc. MFCS Conf., Czechoslovakian Academy of Sciences, 1973.

34] *Hill, A. J.* A predicate logic data base. MSc Thesis, Dept. of Computing and Control, Imperial College, London, 1976.

35] *Hoare, C. A. R.* Algorithm 64. Comm ACM 4, 1961.

36] *Hogger, C. J.* Stepwise refinement for the synthesis of predicate logic programs. Research Report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London, 1975.

37] *Hogger, C. J.* A logic program for the linear programming Simplex Algorithm. Research Report, Theory of Computing Research Group, Dept, of Computing and Control, Imperial College, London, 1976.

38] *Hogger, C. J.* Deductive synthesis of logic programs. Research Report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London, 1977.

39] *Hogger, C. J.* Program synthesis in predicate logic. Proc. AISB/GI Conf. on Artificial Intelligence, Hamburg, July 18-20, 1978.

40] *Hogger, C. J.* Goal-oriented derivation of logic programs. Proc. MFCS Conf., Polish Academy of Sciences, Zakopane, 1978.

41] *Katz, S. and Manna, Z.* Logical analysis of programs. Comm. ACM 19, 1976.

42] *King, J.* A program verifier. PhD Thesis, Carnegie-Mellon University, Pittsburgh, Pa., 1969.

43] *Knuth, D. E.* The art of computer programming. Vol. 3 : Sorting and Searching, Addison-Wesley Publ. Co., 1973.

44] *Knuth, D. E., Morris, J. H. and Pratt, V. R.* Fast pattern matching in strings. Technical Report CS-74-440, Stanford University, Stanford, California, 1974.

45] *Kodratoff, Y.* A sane algorithm for the synthesis of LISP functions from example problems : the Boyer and Moore algorithm. Proc. AISB/GI Conf. on Artificial Intelligence, Hamburg, July 18-20, 1978.

46] *Kowalski, R.* Studies in the completeness and efficiency of theorem proving by resolution. PhD Thesis, University of Edinburgh, 1970.

47] *Kowalski, R.* A proof procedure using connection graphs. DCL Memo No. 74, University of Edinburgh, 1973.

48] *Kowalski, R.* A proof procedure using connection graphs. J. ACM 22 (4), 1975.

49] *Kowalski, R.* Logic for problem solving. DCL Memo No. 75, University of Edinburgh, 1974.

50] *Kowalski, R.* Predicate logic as programming language. Proc. IFIP Congress, 1974.

51] *Kowalski, R.* Algorithm = logic + control. Research Report, Theory of Computing Research Group, Dept. of Computing and Control, Imperial College, London, 1976.

52] *Kuehner, D. G.* Strategies for improving the efficiency of theorem proving by resolution. PhD Thesis, University of Edinburgh, 1971.

53] *Lee, R. C. T. and Waldinger, R. J.* PROW : a step toward automatic program writing. Proc. IJCAI, Washington D.C., 1969.

54] *Liskov, B. and Zilles, S.* Specification techniques for data abstraction. Proc. Conf. on Reliable Software, Los Angeles, California, 1975.

55] *London, R. L.* A view of program verification. Proc. Conf. on Reliable Software, Los Angeles, California, 1975.

56] *Luckham, D. C., Morales, J. J. and Schreiber, J. F.* A study in the applications of theorem proving. Proc. AISB/GI Conf. on Artificial Intelligence, Hamburg, July 18-20, 1978.

57] *McCabe, F. G.* Euclid - a coroutining theorem-prover. MSc Thesis, Dept. of Computing and Control, Imperial College, London, 1976.

58] *McCarthy, J.* Programs with common sense. Mechanization of Thought Processes, Vol. 1, pp. 77-84, Proc. Symp. Nat. Phys. Lab., London, Nov. 24-27, 1958.

59] *McCarthy, J.* A basis for a mathematical theory of computation. In Studies in Logic and the Foundations of Mathematics : Computer Programming and Formal Systems. North Holland Publ. Co., 1963.

60] *Manna, Z. and Waldinger, R. J.* Toward automatic program synthesis. Comm. ACM 14, 1971.

61] *Manna, Z. and Waldinger, R. J.* Knowledge and reasoning in program synthesis. Artif. Intel J. 6 (2), 1975.

62] *Manna, Z. and Waldinger, R. J.* The automatic synthesis of systems of recursive programs. Proc. IJCAI Conf., 1977.

63] *Murray, N.* A proof procedure for non-clausal first order logic. Research Report, University of Syracuse, New, York, 1978.

64] *von Neumann, J. and Goldstine, H. H.* On the principles of large scale computing machines. In The Collected Works of John von Neumann, Vol. 5, Pergamom Press, 1963.

65] *Newell, A., Shaw, J. and Simon, H.* Empirical explorations of the logic theory machine. Proc. West. Joint Computer Conf., Vol. 15, pp. 218-239, 1957.

66] *Newell, A., Shaw, J. and Simon, H.* Report on a general problem-solving program. Proc. Intern. Conf. on Information Processing, pp. 256-264, UNESCO House, Paris, 1959.

67] *Nilsson, N. J.* Problem-solving methods in artificial intelligence. McGraw Hill Book Company, 1971.

68] *Noonan, R. E.* Structured programming and formal specification. IEEE Trans. on Software Engineering, Col. SE-1 (4), 1975.

69] *Quine, W. V. O.* A proof procedure for quantification theory. J. Symbolic Logic, 20 , 1955.

70] *Raphael, B.* SIR : a computer program for semantic information retrieval. PhD Thesis, Mass. Inst. of Technology, 1964.

71] *Reynolds, C. and Yeh, R. T.* Induction as the basis for program verification. IEEE Trans. on Software Engineering, Vol. SE-2 (4), 1976.

72] *Robinson, J. A.* Theorem-proving on the computer. J. ACM 10 , 1963.

73] *Robinson, J. A.* A machine-oriented logic based on the resolution principle. J. ACM 12 , 1965.

74] *Roussel, P.* Prolog : manuel de reference et d'utilisation. University d'Aix-Marseille, 1975.

75] *Russell, B.* International Monthly, p. 85, 1901.

76] *Sandewall, E.* Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. Proc. IJCAI-3, pp. 230-234, 1973.

77] *Schmitz, L.* An exercise in program synthesis : algorithms for computing the transitive closure of a relation. Research Report, Fachbereich Informatich, Hochschule der Bundeswehr, Munich, 1978.

78] *Spitzen, J. and Wegbreit, B.* The verification and synthesis of data structures. Acta Informatica, pp. 127-144, 1975.

79] *Tarnlund, S-A.* An interpreter for the programming language predicate logic. Proc. IJCAI-4, Tbilisi, USSR, 1975.

80] *Tarnlund, S-A.* Unpublished presentation to a Workshop on Logic Programming, Imperial College, London, May 1976.

81] *Wang, H.* Towards mechanical mathematics. IBM J. of Research and Development, 4 , pp. 2-22, 1960.

82] *Warren, D.* A user's guide to the DEC-10 Prolog system. University of Edinburgh, 1975.

83] *Warren, D.* Implementing Prolog, a language for programming in logic. Dept. of Artificial Intelligence, University of Edinburgh, 1976.

84] *Warren, D.* Implementing Prolog - compiling predicate logic programs, Vols. 1 and 2 , DAI Research Reports Nos. 39 and 40, University of Edinburgh, 1977.

85] *Warren, D., Pereira, L. and Pereira, F.* Prolog - the language and its implementation compared with LISP. Proc. SIGPLAN-SIGART Language Conf., Rochester, 1977.

86] *Wegbreit, B.* Goal-directed program transformation. IEEE Trans. on Software Engineering, Vol. SE-2 (2), 1976.

87] *Welham, R.* Geometry problem solving. DAI Research Report No. 14, University of Edinburgh, 1976.

88] *Williams, J. W. J.* Algorithm 232, HEAPSORT. Comm. ACM 7, No. 6, 1964.

89] *Wirth, N.* Systematic programming. Prentice Hall, 1973.

90] *Workshop on Logic and Data Bases.* Toulouse, November, 1977.

---------- E N D ----------