1

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

THÈSES CANADIENNES

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

Canada

TABLE OF CONTENTS

Chapter	Title	Page
2.2.4	Conventional Database Problems	17
2.3	Research Applicability	18
2.4	Conclusions	19
3	Prolog Data Structures and Memory Allocation	20
3.1	Introduction	20
3.2	Data Structures	20
3.2.1	Data Structures Overview	22
3.2.2	The ATOM	34
3.2.3	The FUNCTOR	38
3.2.4	The CLAUSEREC	41
3.2.5	The SKEL	44
3.2.6	The FRAME	45
3.2.7	The MOL	49
3.3	Memory Allocation	49
3.4	Conclusion	52
4	Prolog Interpreter Overview	53
4.1	Introduction	53
4.2	General Operation	53
4.3	Flow of Control	58
4.3.1	The Meta-Level Algorithm	58
4.3.2	The "C" Algorithm	64

LIST OF FIGURES

Figure	Title	Page
1-1	partsdb - User Relations	4
1-2	Widgets	6
1-3	All Widgets	6
1-4	Partsfile	7
1-5	Widget's Search Tree	8
1-6	Interface Overview	11
3-1	"Memory Picture" Symbols	27
3-2	"supplies(foo,pl)" memory picture	28
3-3	"supplies(X,Y):- subpart(Y,Z),supplies(X,Z)"	29
3-4	Heap Memory Picture for supplies/2	30
3-5	Heap Memory Picture for widget/1	31
3-6	Heap Memory Picture for subpart/2	32
3-7	Heap Memory Picture for gadget/1	33
3-8	ATOM Structure Declaration	35
3-9	ATOM Structure Diagram	36
3-10	FUNCTOR Structure Declaration	39
3-11	FUNCTOR Structure Diagram	40
3-12	CLAUSEREC Structure Declaration	42
3-13	CLAUSEREC Structure Diagram	42

Chapter Two

Literature Review

2.1 Introduction

Four naturally occurring subdivisions appear in the literature on logic languages and database management systems. These are:

1. The provision of 'natural language' human/machine interfaces.
2. The use of logic in optimizing database queries.
3. The provision of deductive capabilities to databases or (depending on your viewpoint) the provision of efficient secondary storage access to PROLOG. and
4. The conventional database problems dealing with the provision of security and integrity constraints, ensuring protection of data against software and hardware errors and failures; and the questions of database theory such as the meaning of a null in an attribute of a relation. (Is it zero, a value which exists but is unknown, or a variable for which any value may be substituted?)

The first part of this chapter will attempt to review some of the relevant papers in each of these areas. The review is not intended to be exhaustive but rather to give a feel for the flavour of each area. The second part of the chapter will attempt to show where the topic of this thesis fits in with these areas and how its results may be

PROGRES

AN INTERFACE BETWEEN C-PROLOG AND INGRES

by

E. Arthur George, B.ENG

ABSTRACT

This thesis describes a prototype interface between C-Prolog and the ingres database management system. The prototype permits tuples in a relation in the database to be viewed as prolog ground unit clauses.

The interface is written in C and is incorporated entirely in the code of the C-Prolog interpreter. Two new built-in predicates allow an ingres database to be opened or closed. After opening a database the user treats the tuples of relations as instances of predicates having the same name as the relation.

The Progres system may be viewed as either an ingres database management system with deductive capabilities or as a prolog system capable of accessing a relational database as a source of ground unit clauses.

ACKNOWLEDGEMENTS

I wish to thank the following:

To Professor B. Pagurek, my supervisor, for his constant advice and encouragement through-out this research.

To my wife, Shirley-Ann, for her assistance in proof-reading this thesis and her support during the many long days.

To my children, Jason and Karen, who often put up with my bad temper while I was trying to complete this.

TABLE OF CONTENTS

Chapter	Title	Page
	Abstract	i
	Acknowledgements	ii
	Table of Contents	iii
	List of Figures	vi
	List of Appendices	x
	Glossary of Terms	xi
1	Thesis Introduction	1
1.1	Introduction	1
1.2	Reasons for Interfacing Ingres and Prolog	3
1.3	Implementation Overview	10
1.4	Thesis Overview	10
2	Literature Review	12
2.1	Introduction	12
2.2	Review	13
2.2.1	Natural Language Interfaces	13
2.2.2	Query Optimization	13
2.2.3	Deduction and Databases	15

TABLE OF CONTENTS

Chapter	Title	Page
2.2.4	Conventional Database Problems	17
2.3	Research Applicability	18
2.4	Conclusions	19
3	Prolog Data Structures and Memory Allocation	20
3.1	Introduction	20
3.2	Data Structures	20
3.2.1	Data Structures Overview	22
3.2.2	The ATOM	34
3.2.3	The FUNCTOR	38
3.2.4	The CLAUSEREC	41
3.2.5	The SKEL	44
3.2.6	The FRAME	45
3.2.7	The MOL	49
3.3	Memory Allocation	49
3.4	Conclusion	52
4	Prolog Interpreter Overview	53
4.1	Introduction	53
4.2	General Operation	53
4.3	Flow of Control	58
4.3.1	The Meta-Level Algorithm	58
4.3.2	The "C" Algorithm	64

TABLE OF CONTENTS

Chapter	Title	Page
4.4	Conclusion	79
5	Progres Description	81
5.1	Introduction	81
5.2	Design	85
5.3	Implementation	86
5.3.1	Overview	86
5.3.2	Memory and Data Structure Changes and New Data Structures	95
5.3.3	Process Modifications	98
5.3.4	Meta-Level Modifications	115
5.4	Operation of the Interface	116
5.5	Conclusion	128
6	Analysis and Conclusions	130
6.1	Analysis	130
6.2	Summary and Conclusions	135
	References	138

LIST OF FIGURES

Figure	Title	Page
1-1	partsdb - User Relations	4
1-2	Widgets	6
1-3	All_Widgets	6
1-4	Partsfile	7
1-5	Widget's Search Tree	8
1-6	Interface Overview	11
3-1	"Memory Picture" Symbols	27
3-2	"supplies(foo,pl)" memory picture	28
3-3	"supplies(X,Y):- subpart(Y,Z),supplies(X,Z)"	29
3-4	Heap Memory Picture for supplies/2	30
3-5	Heap Memory Picture for widget/1	31
3-6	Heap Memory Picture for subpart/2	32
3-7	Heap Memory Picture for gadget/1	33
3-8	ATOM Structure Declaration	35
3-9	ATOM Structure Diagram	36
3-10	FUNCTOR Structure Declaration	39
3-11	FUNCTOR Structure Diagram	40
3-12	CLAUSEREC Structure Declaration	42
3-13	CLAUSEREC Structure Diagram	42

LIST OF FIGURES

Figure	Title	Page
3-14	SKEL Structure Declaration	44
3-15	SKEL Structure Diagram	45
3-16	FRAME Structure Declaration	46
3-17	FRAME Structure Diagram	47
3-18	MOL Structure Declaration	48
3-19	MOL Structure Diagram	49
3-20	Prolog Memory Layout	50
3-21	Pseudo Code "lookup" Algorithm	50
4-1	Diagrammatic Representation of User's Goal	54
4-2	Initialization File	60
4-3	Pseudo Code Declarations	65
4-4	Main Control Loop of Pseudo Code	66
4-5	Predicate Resolution	67
4-6	Alternative Instance Handling	68
4-7	Body Goals Handling	68
4-8	Argument Resolution	69
4-9	Built-in Predicate Evaluation	70
4-10	Goal Tracking	70
4-11	Continuation Handling	71
4-12	End-of-goal Handling	71
4-13	Instance Failure Handling	72
4-14	Goal Failure Handling	73

LIST OF FIGURES

Figure	Title	Page
4-15	Global Frame for User-goal subpart(w5,g2)	74
4-16	Global Frame for User-goal subpart(X,g2)	78
5-1	Heap And Relstack Entries For supplies/2	89
5-2	Heap And Relstack Entries For subpart/2	90
5-3	Heap And Relstack Entries For widget/1	91
5-4	Heap And Relstack Entries For gadget/1	92
5-5	ATTRIBUTE Structure Declaration	96
5-6	ATTRIBUTE Structure Diagram	97
5-7	EXIT_Entry Declaration	97
5-8	EXIT_Entry Structure Diagram	98
5-9	Retrieving Qualifying Tuples Into A Temporary Relation	100
5-10	Retrieving Tuples From Temporary	101
5-11	Pseudo Code For Opening A Database	102
5-12	Pseudo Code For Exiting A Database	103
5-13	Removing Temporarary Relations From The Database	104
5-14	Creating The Database Environment	105
5-15	Removing The Database Environment	106
5-16	Qualifying String For Retrieve Into Temporarary	107

LIST OF FIGURES

Figure	Title	Page
5-17	Build FUNCTOR's And ATOM's For Database Relations	108
5-18	Modified Initialization File	109
5-19	Main Control Loop Modified For Interface	111
5-20	Modified Predicate Resolution Procedure	112
5-21	Test_Alternatives (Flow Control) Procedure	113
5-22	Modified Alternate Instance Handling	113
5-23	Modified Instance Failure Handling	114
5-24	Modified Goal Failure Handling	115
5-25	Selected Parts of partsdb System Relations	117
5-26	Rules_File	122
6-1	Reversed Retrieval Ordering	131

LIST OF APPENDICES

Appendix A

Prolog Initialization File

GLOSSARY OF TERMS

Term	Definition
argument	comma separated terms within the brackets of a compound term (see compound term)
Note	Because the arguments are terms, they may themselves be compound terms. eg. grandmother(mother(X,Y),mother(Y,Z)).
arity	count of number of arguments in a compound term
atom	any sequence of characters which cannot be mistaken for any other prolog symbol or which is quoted (single quotes). eg. betty, mother, ancestor, '23', [].
attribute	column of a relation or, in a tuple, the data in the position corresponding to the column of the relation from which the tuple is drawn.
clause	analogous to an English sentence, a clause has the sense of "(head) is true if (body) is true." or "To prove (head) is true, prove that (body) is true." In Prolog the head is either a single literal or is empty, while the body has zero, one or more literals connected by conjunctives (',') or disjunctives (';'). There must be at least one literal in a clause.

GLOSSARY OF TERMS

Term	Definition
command	a headless clause starting with "?-" or ":-". It requires prolog to prove the command's truth or falsity. Also called a negative clause.
compound-term	consists of a functor (an atom called the principal functor) and one or more terms in brackets called the arguments.
conjunction	logical connection between two objects. In logic, the conjunction has the meaning "AND". The comma (,) signifies a conjunction in prolog. (See also disjunction)
constant	a discrete, non-capitalized, or quoted object. eg. 1, mother, alice, 'Bob'.
consult	When prolog "consults" a file, it reads it. While the file is being read, commands are executed and all other clauses are stored in memory as facts or rules.
C-Prolog	a prolog interpreter written in the language "C".
deduction	Funk and Wagnels. defines deduction as reasoning from stated premises to conclusions formally or necessarily implied by such premises. Prolog automates this process.

GLOSSARY OF TERMS

Term	Definition
disjunction	logical choice. In prolog disjunction means "or" and is symbolized by the semi-colon (;).
functor	generic descriptor applying to a set of clauses which have the same name and number of arguments. e.g. "supplies(foo,pl) and "supplies(acme,X) :- widget(X)" are both members of the supplies/2 functor.
goal	logical conclusion prolog attempts to prove from its facts and rules (premises).
ground unit clause	- a clause with only a head (no body literals) and whose arguments are constants. See also: clause, non-unit clause, command.
instance	a specific case or example of a clause.
ingres	a (relational) database management system
invocation	the act of causing a particular fact or rule to be used in proving a goal.
literal	the fundamental unit of a logic program, the literal is a compound term which has a predicate as its principal functor. The only means generally available to distinguish a literal from any other compound term is its context.
negative clause	see command.
non-unit clause	A clause having both a head and a body (ie a rule).

GLOSSARY OF TERMS

Term	Definition
operators	<p>Specially declared functors having the following properties:</p> <ol style="list-style-type: none"> 1. Has either one or two arguments. 2. Has a "precedence" for determining order of evaluation with respect to other operators. 3. Has a "type" used to determine order of evaluation in the presence of other operators having the same precedence and for determining whether it is a unary or binary op.
predicate	essentially the same as the principal functor of a compound term but, as in English grammar, it has the sense of an action or adjective. eg. ancestor(X,Y) as in X is an ancestor of Y.
primitive	in prolog, a number (either an integer or a floating point number)
principle functor	see compound term
project	<p>in ingres, a vertical selection from a relation. e.g.,</p> <p>On a retrieval from a relation, a projection would cause only some of the attributes to be displayed.</p>
relation	conceptually, a named table of rows and columns. Only columns (attributes), within the table, are named. See also: attribute, tuple.
rule	<p>Equivalent to a logical premise, a rule may be read as</p> <p>"(head) is true if (body) is true". See also: clause, non-unit clause:</p>

GLOSSARY OF TERMS

Term	Definition
session goal	The goal which C-Prolog constantly attempts to prove. A side effect is that user's commands are executed and goals solved. When the user enters "end_or_file", this goal succeeds and C-Prolog exits.
select	in ingres, causes a display of only those rows of a relation meeting certain qualifications.
term	A term is any identifiable discrete data object. This includes constants, variables and compound terms. eg. anne, X, father(alfred,bob).
tuple	one row of a relation.
variable	an unquoted capitalized object. For prolog purposes the underscore character is a capital letter. The underscore used alone is an unnamed variable (anonymous) which may be used as a place-holder for arguments about whose value we have no interest. eg. X, My_name, _not_anonymous.
Note	Variables in Prolog are not writable storage locations as in other programming languages but rather the name of some data object whose scope is strictly limited to the clause in which it occurs.

Chapter One

Thesis Introduction

1.1 Introduction

Over the last several years much attention has been given to so called artificial intelligence and expert systems. Examples of languages which have been created for this purpose are LISP and PROLOG. These rely on list processing and first order predicate calculus respectively for their underlying support structure. Applications such as MYCIN have shown that there is potential for such language systems to be extremely useful but have also shown that, in general, there is a requirement for greater efficiency and larger memory or a means of efficiently accessing secondary storage.

A second area that has received attention is the management of very large data bases. While much has been done and written about ensuring security, integrity and recovery protection, almost nothing has been done to provide the end user with an interface which is simple to use and which at the same time is flexible. In addition there seems to have been little progress in providing deductive capabilities to a database management system.

This thesis describes an interface between the C-Prolog interpreter written by Fernando Pereira and the ingres relational database management system written by M. Stonebraker et al. A prototype has been completed which permits tuples stored in an ingres database relation to be viewed as though they were prolog ground unit

clauses (see glossary). The system is called PROGRES (Prolog and ingres). The prototype does not implement any of the ingres commands as the goal of this research was only to show that ingres tuples could be used as PROLOG facts.

Daniel Fishman[4] and David Majert[11] have both shown that there is a close similarity between PROLOG's ground unit clauses {predicate_name (arg1, arg2,...)} and a relational database relation {relation_name (attribute1, attribute2,...)}. They conclude that such a similarity suggests that PROLOG and a relational database management system could be unified to provide PROLOG with a secondary database of ground unit clauses. Their analysis gives three ways to provide this marriage:

1. provide database management capabilities to the PROLOG interpreter:
2. provide PROLOG like deductive capabilities to the front end of a relational database.
3. create an interface between the PROLOG interpreter and the dbms such that they work as separate but mutually supporting entities.

PROGRES is essentially an implementation of a mixture of the second and third alternatives given above. Once the user gives the command to the interpreter to open the database, it silently accesses relational tuples as though they were PROLOG facts. Note: PROGRES only does deductive retrievals on the database. The two systems do work

independently, in the sense that either can be invoked without reference to the other, and PROGRES does not access the database relations directly but invokes ingres to retrieve data as necessary.

Detailed and current information is rapidly becoming the life blood of business and industry. The challenges in obtaining, storing, and retrieving this information, so that intelligent decisions can be made, are being addressed in many different ways. Business in general, since the advent of affordable, powerful mini and micro computers, has focused much of their attention on software which assists them in meeting those challenges. PROGRES has the potential of allowing businessmen and their staffs, access to information in a way which lets them, not only view the data in an organized way, but also to extrapolate new information which is directed towards the specific problem they are trying to solve.

1.2 Reasons for Interfacing Ingres and PROLOG

Both ~~deductive interpreters~~ such as C-PROLOG and database management systems like ingres fulfill their proported uses. C-PROLOG does permit the user to derive conclusions from a limited set of basic facts and rules. Ingres does securely, reliably, and reasonably quickly store and retrieve data and it permits data to be viewed in alternative ways.

However, as the following example shows, ingres has a limitation. Ingres cannot use a rule (premis) to deduce facts which are not explicitly stated in the data. An interface to a deductive

interpreter would provide this facility.

Suppose we have the ingres database of figure 1-1. Suppose further, we have the following rules about the data in the database:

- 1. that the company "acme" supplies anything which is a widget.,
- 2. that suppliers supply anything which is a subpart of anything which they supply., and
- 3. that anything that is a subpart of a gadget is a widget.

FIGURE 1-1 - partsdb - USER RELATIONS

Now, we ask ingres for a list of widgets. In Quel, the ingres query language, this request is:

"range of w is widget"

"retrieve (w.all)"

The response is a reproduction of the widget relation from figure 1-1. Ingres has no knowledge of the widgets and gadgets rule and, therefore, cannot deduce that "w4", "w5" and "w6" are widgets even though people can deduce that by just mentally applying the rule to the database. To obtain this information from ingres requires that the rules be applied manually as follows.

"range of w is widget"

"range of g is gadget"

"range of s is subpart"

"retrieve (s.subprt)

where s.subprt == g.gpart"

"retrieve (w.all)"

This manual application of the rule provides the names of all widgets in two relations as shown in figure 1-2. To get the information into a relation such as the "all_widgets" relation of figure 1-3, it would first have to be created as an image of the widgets relation. Then the remaining widgets would have to be appended one at a time. Clearly, this would be far too laborious for a complex set of rules, a complicated query, or a large database. Ingres does allow one to write programs to automate the processing of queries. However, suppose there are changes in the form of the data, in the information required, or in the rules about the data. Any of these changes can quickly make these programs obsolete.

Similar difficult operations would have to be made to retrieve the widgets that "acme" supplies. Simply asking for the objects "acme" supplies would return only "p4" when it should show all the widgets as well. This implies that ingres "needs" a deductive capability.

FIGURE 1-2 - WIDGETS

all_widgets

w1
w2
w3
w4
w5
w6

FIGURE 1-3 - ALL_WIDGETS

Figure 1-4 shows the clauses of a PROLOG version of the ingres database of figure 1-1. In addition, it shows the rules described above written as PROLOG clauses (last three lines). Suppose this information is in a file (partsfile) and C-PROLOG "consults" the file (it reads the facts and rules into memory).

supplies(foo,p1).	i1
supplies(bar,p1).	i2
supplies(acme,p4).	i3
supplies(foo,g1).	i4
subpart(w4,g1).	i5
subpart(w5,g2).	i6
subpart(w6,g2).	i7
subpart(p2,p1).	i8
subpart(p3,p1).	i9
subpart(w1,p4).	i10
subpart(w2,p4).	i11
subpart(w3,w2).	i12
gadget(g1).	i13
gadget(g2).	i14
gadget(g3).	i15
gadget(g4).	i16
widget(w1).	i17
widget(w2).	i18
widget(w3).	i19
supplies(acme,X) :- widget(X).	r1
supplies(X,Y) :- subpart(Y,Z), supplies(X,Z).	r2
widget(X) :- subpart(X,Y), gadget(Y).	r3

FIGURE 1-4 - PARTSFILE

Suppose the PROLOG goal, "widget(2), print(Z), nl, fail." were entered (find an object which is a widget and print it). The "fail" goal will cause this action to repeat until all widgets are listed. PROLOG will return "w1" through "w6" as successive answers. Ignoring the print, newline, and fail goals, the process that PROLOG goes through is a search tree as shown at figure 1-5. The letter/number combinations at the right in figure 1-4 are used in the tree to specify the rule or instance being used at a given point in the tree.

The interpreter first matches on the basic facts producing successes 1, 2, 3. Then it matches the head of rule 3 and attempts to satisfy the goals in the body of the rule. The deductive process

FIGURE 1-5 - WIDGET'S SEARCH TREE

produces successes 4, 5, and 6. Further attempts to satisfy the body goals, shown at the bottom right of the search tree, fail to produce any more parts which are subparts of gadgets.

Because the original goal had a subgoal, "fail", the entire user

goal fails eventually, but in the process all the widgets are found and printed out. This shows that the ability to add a rule to the database permitted the user to extract all the implicit information about widgets by using a single, simply defined query.

Clearly, PROLOG can deduce more information from a database than explicitly appears there. Why not simply use PROLOG to manipulate the data? There are two reasons. Ingres and other commercially available relational database management systems are already being used to support large databases. They have properties such as integrity, security, and error protection facilities built-in which PROLOG does not support. We, therefore, need to provide the deductive capability to the database management system.

There were several reasons for choosing C-PROLOG and ingres as the program vehicles for this research, not the least of which, was that source licenses for both are held by the Systems and Computer Engineering department at Carleton University. Both the C-PROLOG interpreter and ingres are written in "C" and since a very tight coupling of the programs was anticipated, a common language was essential. Ingres offered still another advantage in its Equel (Embedded Quel) pre-compiler. Equel processes a "C" program containing embedded Quel statements and outputs the program with the Quel statements transformed into calls on library functions. The library functions support interaction between the program and ingres.

1.3 Implementation Outline

Briefly, PROGRES was created by modifying the C-PROLOG interpreter to provide two new built-in predicates (opendb/1, exitdb/0) and by introducing several new routines which allowed PROGRES to view relation tuples as though they were PROLOG ground unit clauses. Ingres was not modified. The actual interface was created in "C" using embedded ingres QUEL commands which are converted to calls on library functions by the Equel pre-compiler. These library functions are provided as part of the ingres package. Figure 1-6 shows a high level view of the interface. The interface uses temporary relations to hold tuples of a relation whose attributes are to be used as the arguments of a PROLOG goal. A detailed description of the implementation is contained in chapter V.

1.4 Thesis Overview

The next chapter is an overview of a segment of the current literature in the fields of database management and deductive languages. Particular attention is given to currently recognised problems in these fields with a view to showing how the product of this research may provide tools for solving some of those problems.

One of the most difficult problems encountered in this research was gaining an understanding of the data structures and control flow of the C-PROLOG interpreter. C-PROLOG is a translation of the original program written by Luis Dumas in the language "IMP". Because IMP treats addresses as integers, Fernando Pereira took certain liberties in his translation to overcome "C"'s strong typing

characteristics. This makes the code extremely difficult to read and the overall structure and purpose of the code is almost impossible to discern. Chapter III, therefore, describes in detail the data structures used in C-PROLOG and chapter IV utilizes detailed pseudo code to describe the internal operations of the unification algorithm.

Chapter V describes the actual modifications made to the interpreter code and data structures. Practical problems had to be solved regarding tuple access and storage algorithms, memory allocation, and user views. These problems and their solutions are also described.

Further modifications to the interpreter should be made in order to make it a complete tool for accessing and updating databases and to permit more functional analyses of whether this approach to the provision of deductive capabilities to database management systems is practical. These modifications and a summary of this research are detailed in chapter VI.

FIGURE 1-6 - INTERFACE OVERVIEW

Chapter Three

PROLOG DATA STRUCTURES AND MEMORY ALLOCATION

3.1 Introduction

As shown in chapter I, prolog matches facts to goals in a recursive depth first, left to right search pattern (figure 1-5). To do this, it uses chains (actually linked lists) of data structures. These serve to guide the control processes to the facts and rules, which might unify with a user's goal. The unification algorithm then attempts to match the arguments of the goal with the arguments of the first fact or rule. If unification is successful, the control process then checks for more goals (in a rule) which must be solved. If there are none, then the original goal is proven. If unification with a fact is not successful, succeeding facts or rules are tried until either a successful match is made or there are no more facts (rules) to attempt. In the latter case, the original goal is considered false. This process of trying to match successive facts can be likened to reading successive records in a file until the key items in the record match those being searched for.

The Progres system operates in exactly the same fashion except that there are facts which are specially flagged as "ingres facts". The ingres facts trigger the interface, causing the ingres database to be accessed. The processes of the interface retrieve, into a temporary relation, those facts which will match with the user's goal. (To guarantee matching retrieved facts to the user's goal, arguments in the user's goal which are constants are used to qualify the

applied to each of the areas.

2.2 Review

2.2.1 Natural Language Interfaces

J.C.Neves, S.O.Anderson and M.H.Williams [13] have built an human/machine interface using PROLOG which emulates the Query-By-Example (QBE) system developed by M.M.Zloof. In their design, PROLOG ground unit clauses are considered equivalent to tuples of a relational database. They state that they had little difficulty implementing this kind of database query system in PROLOG.

D.H.D.Warren and F.C.N.Pereira [16] have used PROLOG to interpret natural language queries. They have used this as an interface to their CHAT-80 PROLOG database management system which incorporates a query optimization algorithm.

The discussions in these papers make it clear that the creation of a suitable human/machine interface is relatively straight forward in PROLOG.

2.2.2 Query Optimization

R. Demolombe's [3] paper presents a theoretical statistical model by which one can estimate the cardinality (number of facts to be examined) of a query and any subqueries. The abstraction of the cardinality of subqueries permits a query ordering strategy to give the fewest intermediate retrievals for a given query, which is, in effect, a 'least cost' strategy.

J. Grant and J. Minker [7] describe an algorithm for optimizing queries to a database once a query has had all implications (fact generating rules) reduced to explicit table queries. This optimizer gives a strategy whose cost factor is in terms of 'least time' to produce a result. It achieves this by determining all tables which must be accessed and then using select and join to achieve a query strategy requiring the fewest returned tuples.

E.P. Stabler Jr. and E.W. Elcock [15] describe a number of strategies for preprocessing a PROLOG query, before execution, to help ensure an efficient query ordering. These include indexing the database, goal selection based on the number of possible results to a given subgoal, and selective back tracking. Their approach is pragmatic in the sense that no new predicates need be "built-in" in PROLOG. They simply build the optimizer in PROLOG on top of the interpreter. These strategies would have to be 'fine tuned' for the particular database and applications.

M. Hammer and S.B. Zdonichi, Jr. [8] take a somewhat different approach to query optimization. They suggest that as database management system become more of a tool in the hands of unsophisticated users, queries which are unsophisticated and at the same time difficult to answer by straight forward query optimization are more likely to occur. As an example they suggest a database which contains data about employees according to the departments in which they work. If a query requesting the names of all employees who have worked for the company for more than five years is asked, a brute

15
force method would require that every record be examined. If, however, a query optimizer has knowledge about the database and "knows" that half the departments are less than 5 years old and that employees stay in the same department throughout their careers, then it can deduce that only the other departments employee records need be examined. They suggest a "Knowledge Based Query Processor" based on knowledge about the database application.

PROLOG can be used for both the traditional query optimizer which seeks an efficient query based on knowledge about the database and for knowledge based optimizers which are application sensitive. Both need to be able to "deduce" what an optimal query "should" be based on an arbitrary input query.

2.2.3 Deduction and Databases

J.Chomicki and W.Grudzinski [1] describe a database support system for PROLOG which is essentially an augmented PROLOG (extra commands such as `create_relation()`, `insert_tuple()`, etc) with a separate database system. This system explicitly supports recursive calls (rules which have subgoals which call the rule) and does partial matching on the database for the PROLOG system. Partial matching means that, if some arguments of a ground unit clause have been instantiated when the clause is invoked, the database system will retrieve tuples whose arguments match those facts. The database system does not support consistency checking directly (This can be done at the PROLOG level but not for multiple concurrent users) nor

does it support security or integrity checking constraints. This system may be thought of as a combination of the first and third methods discussed in Chapter 1. The database management system and PROLOG are separate entities (third method) but the database management system was built specially to support PROLOG's recursive calls. Therefore, the system cannot really be thought of as a PROLOG interpreter utilizing an arbitrary database management system as its source of ground unit clauses.

V. Dahl [2] has created a special front end in PROLOG which treats PROLOG ground unit clauses as essentially relational database management system tuples and thus gives the flavour of a deductive relational database without really creating one. With this system, query optimization and other database techniques can be tested in a deductive atmosphere.

I. Futo, F. Darvis, and P. Szerdi [5] describe an extended PROLOG which manipulates "relation like" files. It has been used for a drug data and drug interactions question/answering system and for pesticide and air pollution database systems. They found their work to be directly applicable and draw the conclusion that PROLOG was an easy language to manipulate so that working structures could be developed as the total structure developed. They further concluded that PROLOG should be used for "query, reply, and control" while traditional file handling systems should be used for data retrieval.

C. Kellog and L. Travis [10] discuss Deductively Augmented Data Management (DADM) in the context of applying the theory of logical

deduction and proof procedure to the accessing of data stored in conventional database management system. They cover many of the aspects of the four areas outlined at the beginning of this Chapter. They explicitly state that their deductive processor is deliberately held separate from the database management system rather than embedding one in the other in order to take advantage of emerging database management systems.

J. Minker [12] describes an experimental database system whose founding structure is predicate calculus. He reasons that logic is a useful way in which to represent knowledge and provides a mathematical basis for reasoning with data and for maintaining data integrity. The thrust of the paper is directed to showing how current database problems may be solved using logic.

K. Parsaye [14] shows that some relatively innocuous extensions to PROLOG can serve to create a convergence of many concepts associated with relational database management and knowledge base management.

2.2.4 Conventional Database Problems

H. Gallaire [6] surveys the current problems in database management system and shows how logic may be used as a formal support for their study and further how it may be used to solve them. Among the problems enumerated are:

1. the embedding of a "natural language" interface in a database management system,

2. null values meaning "attributes applicable but value unknown",
3. integrity constraints - domain type, dependency and transactions,
4. query optimization, and
5. negation as failure.

T. Imielinski and W. Lipski, Jr. [9] provide several systems which are safe and complete for use with subsets of the allowable database operators and databases containing null values. In this context, 'safe' means the system will not give an answer which is wrong, and 'complete' means that the system will find all possible solutions to a problem. Their results do not comprehensively solve the null value problem but show promise of at least a partial solution.

2.3 Research Applicability

The review above has shown that logic and logic languages show promise of being useful in assisting in the solution of many problems in database practice. This suggests that a tool incorporating a logic interpreter as the man/machine interface to a database might be a very practical object.

Once the tool is built, extremely powerful natural language interfaces can be built by storing facts in the database with rules about the facts stored in memory. This can, in turn, be enhanced by the addition of logical query optimization. In fact, if one approaches the problems of natural language interfacing and query optimization from the viewpoint of Hammer and Zdonicki [8] where the

particular application is important to the query optimization (Knowledge Based Query Processing), then the syntactic processing becomes part of the query optimization process.

The tool, itself, is a solution to the problems of the third area noted at the beginning of the chapter (ie. the provision of deductive capabilities to a relational database management system):

Other database problems, as Gallaire [6] has shown, such as the null value problem should be amenable to solution using logic. The provision of a deductive front end to a database management system could provide the tool by which these problems are made soluable.

2.4 Conclusion

Based on the literature, there is a requirement to provide deductive capabilities to a database management system. As shown in chapter one, a feasible method of providing these capabilities is to devise an interface between a deductive interpreter such as C-PROLOG and a relational database management system such as INGRES. This has benefits in both the realms of deductive interpreters (provision of access to large stores of current facts) and database management systems (natural language interfaces, query optimization, conventional database theory problems).

Chapter Three

PROLOG DATA STRUCTURES AND MEMORY ALLOCATION

3.1 Introduction

As shown in chapter 1, prolog matches facts to goals in a recursive depth first, left to right search pattern (figure 1-5). To do this, it uses chains (actually linked lists) of data structures. These serve to guide the control processes to the facts and rules, which might unify with a user's goal. The unification algorithm then attempts to match the arguments of the goal with the arguments of the first fact or rule. If unification is successful, the control process then checks for more goals (in a rule) which must be solved. If there are none, then the original goal is proven. If unification with a fact is not successful, succeeding facts or rules are tried until either a successful match is made or there are no more facts (rules) to attempt. In the latter case, the original goal is considered false. This process of trying to match successive facts can be likened to reading successive records in a file until the key items in the record match those being searched for.

The Progres system operates in exactly the same fashion except that there are facts which are specially flagged as "ingres facts". The ingres facts trigger the interface, causing the ingres database to be accessed. The processes of the interface retrieve, into a temporary relation, those facts which will match with the user's goal. (To guarantee matching retrieved facts to the user's goal, arguments in the user's goal which are constants are used to qualify the

retrieval). The first fact/tuple in the temporary is brought into memory and unified with the user's goal. To keep this fact from being retrieved again, should a later user's goal fail, it is now deleted from the temporary relation. The process now proceeds to solve the other user's goals (if any). The temporary relation is retained until either all the user's goals are successfully solved, or it is emptied as a result of repeated user goal failures. If the temporary relation is emptied, then the "ingres fact" has failed and Progres tries to unify with any other facts or rules in the same way that prolog does.

The key element in the above, very broad, description of prolog/Progres is the data structures. These contain the information which the control processes need in order to deduce correct conclusions concerning the user's goals. Also relevant is the way memory space is allocated and used. These two topics are the basis for the discussions in the rest of this chapter. Chapter IV will discuss the control processes.

3.2 Data Structures

As mentioned in chapter I the original Prolog interpreter was written in IMP and treated addresses as integers. The 'C' compiler enforces strong typing which does not directly permit this freedom. To overcome this disparity in the languages, Fernando Pereira relies heavily on the 'C' 'typedef' and 'type casting' operators. One type definition is critical to an understanding of the data structures. 'PTR' is an object declared as follows:

```
typedef unsigned **PTR;
```

That is, PTR is a pointer to a pointer to an unsigned integer. Aside from the basic 'C' objects (char, int, etc), virtually all objects in C-Prolog are of this type or are structures built from this type. Structures are typedef'd and addressed using typecast PTR's. Do not be confused by the PTR declaration. Think of it strictly as an object which is used to point to another object. The reason for mentioning it, is that its usage in the C-Prolog code is confusing until one learns to think of it as described above.

3.2.1 Data Structures Overview

There are six structures which play a central role in the operation of the interpreter. They are:

ATOM,

FUNCTOR,

CLAUSEREC,

FRAME,

SKEL, and

MOL.

These structures are associated with various prolog objects. The "facts" referred to in the introduction are a combination of the CLAUSEREC and SKEL structures. In the following discussions, pay particular attention to these structures, their internal pointers, how they are connected to each other, and how they are connected to other

structures. The modifications made to these structures is central to the Progres implementation.

Returning to the prolog database of figure 1-4, all the "words" are atoms and are stored in ATOM structures. For example, "supplies", "w4", and "gadget" are all stored as strings within separate ATOMs.

An atom which is also a principle functor (predicate name) will have one or more FUNCTOR structures associated with it. This association is created by placing a pointer to the ATOM in the FUNCTOR structure. A given atom may be the principle functor for clauses having differing numbers of arguments. For instance, in figure 1-4, the "supplies" clauses all have two arguments. This is called their arity. But there could have been supplies clauses having 1 or 3 or any number of arguments (up to 255). There is a pointer in the ATOM structure which points to the first FUNCTOR in a chain of FUNCTORS, all of which have that atom as their principle functor, but which are associated with clauses having differing arities. These FUNCTORS will be identified by referring to them as "predicate name/arity". For example, all the "supplies" clauses in figure 1-4 are associated with the FUNCTOR "supplies/2" and all the "gadget" clauses are associated with the FUNCTOR "gadget/1".

A FUNCTOR structure is used as a descriptive header for clauses associated with it. It does not, by itself, describe a particular clause. Instead it contains pointers to the beginning and end of a chain of CLAUSERECs and other pertinent information.

There is a CLAUSEREC associated with every clause in the

database. The CLAUSEREC does not directly contain any information about the specific clause. Instead, it contains pointers to the CLAUSERECs before and after it in the chain, information about the number and type of variable arguments, flags indicating the clause's status (in use, erased) and (a) pointer(s) to (a) SKEL structure(s). The CLAUSEREC therefore, serves as a kind of signpost or information desk.

The SKEL structure is where the "real" information about a clause is stored. The SKEL(eton) contains a pointer to a FUNCTOR plus other information. Before describing this information it should be noted that numbers (integers and floats) in prolog are specially constructed values. That is, they are deliberately made to be negative and their interpretation as numbers is handled exclusively by C-Prolog. These numbers are referred to as "primitives". As well, variables are identified specially. Remember, that variables in prolog are not writable storage locations but rather the name of some data object whose scope is strictly limited to the clause in which it occurs. Primitives, variables, and pointers all occupy one word of memory.

The remaining information in the SKEL structure is a series of memory words corresponding to the arguments of the clause with each word containing either a primitive, a variable, or a pointer. The SKEL for "supplies(foo,pl)" would contain pointers to supplies/2, the ATOM for "foo", and the ATOM for "pl". If there was a clause like "supplies(bar, gadget(X))" then the SKEL would contain pointers to supplies/2, the ATOM for "bar", and a pointer to another SKEL

containing a pointer to gadget/1 and a variable indicator. The combination of a CLAUSEREC and associated SKEL or SKELs will often be referred to as an "instance". The combination can be thought of as the "record header" (CLAUSEREC) and "record" (SKEL) of a sequential file.

If the clause associated with a particular CLAUSEREC is a rule, it is handled in much the same way. Let us use the rule "supplies(acme,X) :- widget(X)." The "head" of this rule is "supplies(acme,X)" so the CLAUSEREC for it will appear in the chain of CLAUSERECs associated with supplies/2. The CLAUSEREC will have a pointer to a SKEL containing a pointer to supplies/2, a pointer to the ATOM for "acme" and a variable indicator. A second pointer in the CLAUSEREC will point to the "body" (widget(X)) of the clause. In our example, it will point to a SKEL containing a pointer to widget/1 and a variable indicator.

What happens if the "body" contains more than one goal? To explain that, let us use the second supplies/2 rule, "supplies(X,Y) :- subpart(Y,Z), supplies(X,Z)." There is a built-in predicate in prolog called the comma functor (,/2) which takes two arguments and has the meaning "AND". That is, the above rule says "X supplies Y" if "Y" is a subpart of "Z" AND "X" supplies "Z". To make clear how prolog interprets this, let us rewrite the rule as "supplies(X,Y) :- ,((subpart(Y,Z)),(supplies(X,Z))).". The "body" pointer in the CLAUSEREC would, then, point to a SKEL which contained a pointer to ",/2" and pointers to two other SKELs. The one SKEL would contain a pointer to subpart/2 followed by two variable indicators and the other

SKEL would have a pointer to supplies/2 followed by two variable indicators.

Figure 3-1 shows the symbols which will be used to provide "memory pictures" of the data structure associations. Figure 3-2 shows the memory picture of "supplies(foo,pl)" and figure 3-3 shows the second supplies/2 rule. Figures 3-3 through 3-7 show the complete picture of the FUNCTOR, CLAUSEREC and SKEL structures of the database of figure 1-4. The ATOM structures are not shown in figures 3-4 to 3-7. Where atoms appear in the SKELs, remember these are pointers to ATOMs or FUNCTORs and are not the strings themselves.

ATOM/FUNCTOR STRUCTURE

CLAUSEREC STRUCTURE

SKEL STRUCTURE

FIGURE 3-1 - MEMORY PICTURE SYMBOLS

FIGURE 3-2 - 'supplies(foo, pl)' MEMORY PICTURE

FIGURE 3-3 - MEMORY PICTURE FOR
`supplies(X,Y):-subpart(Y,Z),supplies(X,Z).`

FIGURE 3-5 - MEMORY PICTURE FOR 'widget/1'

FIGURE 3-6 - MEMORY PICTURE FOR 'subpart/2'

FIGURE 3-7 - MEMORY PICTURE FOR 'gadget/1'

FRAMES and MOLs are not data structures in the same sense as the other structures. Instead, they might be more accurately termed to be control structures.

A FRAME is built whenever a new goal is encountered. It contains pointers to the address of the next instance (it points to the `altofcl` pointer in the current `CLAUSEREC`), the FRAME of the "parent" goal (ie the rule that caused this goal to be invoked) and other information that must be retained so that the interpreter can determine which goal to try next if there is a failure to unify on this instance or on some later goal.

MOLs are used to provide pointers to user goals for the control

processes. More details on the use of FRAMES and MOLs will be given when the control processes is described in chapter IV.

The following detailed descriptions of the structures is intended to help show the connections between these structures. A description of how these linkages are used to solve a user's goals will be given in chapter IV.

3.2.2 The ATOM

Figure 3-8 is a reproduction of the declaration of the ATOM structure. Included in the figure is a number of definitions which were used to reduce the complexity of the structure's use in the writing of the actual program. Figure 3-9 is a diagrammatic representation of the structure.

The numbers in figure 3-9 indicate the number of bytes occupied by each object. The objects are described below.

```

/* atom entry */

#define infofae  AInfo.Infofae
#define arityofae AInfo.ABytes.Arityofae
#define flgsofae AInfo.ABytes.Flgsofae
#define infixofae AInfo.ABytes.InfixOfAE
#define defsofae AClauses.First
#define dbofae   ARecords.First

typedef struct ATOM {
    PTR      atofae; /* self pointer */
    union {
        unsigned Infofae; /* information */
        struct {
            char  Arityofae; /* arity ( = 0 ) */
            char  Flgsofae; /* flags field */
            short InfixOfAE; /* infix priority */
        } ABytes;
        } AInfo;
    Header AClauses; /* chain of definitions */
    Header ARecords; /* chain of records */
    PTR    fcofae; /* functor chain */
    PTR    nxtofae; /* hash chain */
    short  prfxofae; /* prefix and
                    psfxofae; /* postfix priorities */
    unsigned char I_flag;
    char  stofae[1]; /* string for atom */
} ATOM;

```

FIGURE 3-8 - ATOM STRUCTURE DECLARATION

FIGURE 3-9 - ATOM STRUCTURE DIAGRAM

atofae: atom of atom entry - a self pointer. That is, it contains its own address. Its existence is directly attributable to the fact that C-Prolog treats ATOM and FUNCTOR entries in an almost identical manner and requires an object to point to which contains the address of the containing structure. The algorithm also requires that the first entry of both structures point to its associated ATOM.

AInfo: Atom Information - This union is necessitated by the requirement to keep the FUNCTOR and ATOM structures identical in form.

ABytes: Atom Bytes - Information structure.

Arityofae: arity of atom entry - This entry is always zero since, by definition, an atom cannot have arguments.

Flgsofae: Flags of atom entry - used to ensure that built-in predicates are not traced, that they cannot be redefined, etc.

Infixofae: Infix of atom entry - If this atom is an infix operator (*,<,/,etc), then its priority (1-1200) will be stored here.

AClauses: These objects are both structures defined as follows:

```

ARecords: typedef struct {
    PTR First;
    PTR Last;
    Header;

```

In an atom entry, AClauses.First may contain a number representing a case label in a switch statement. The instances of this are fairly rare and encompass built-in predicates which have no arguments such as !(cut), nl, abort, etc where the predicates cannot have arguments. The case label is used to select a code segment which accomplishes the goal of the built-in predicate. In other cases, these two structures may contain NULL's or addresses of CLAUSEREC,s (described later).

fcofae: FUNCTOR chain of atom entry - This pointer, if not NULL, points to a chain of functor entries whose predicate name is this-atom.

nxtofae: next of atom entry - atom entries are accessed via a hash table. All entries hashing to the same point in the table are chained together by this pointer.

prfxofae: prefix of atom entry.

psfxofae: postfix of atom entry .

priorities of prefix and postfix operators. Note: some operators have more than one priority. For example, "+" and "-" both have both prefix and infix priorities, and "." has both prefix and postfix priorities.

stofae: string of atom entry - this is the actual string which is the atom. The declaration is somewhat misleading. An atom may be up to 255 characters long. Because the interpreter does its own memory management, the string is copied to the address of stofae and memory pointers for the next atom entry are bumped past the end of the string.

3.2.3 The FUNCTOR

Figure 3-10 is a reproduction of the FUNCTOR structure declaration and figure 3-11 is a diagrammatic representation of the structure.

```
/* functor entry */

#define infoffe      FInfo.Infoffe
#define arityoffe    FInfo.FBytes.Arityoffe
#define flgsoffe     FInfo.FBytes.Flgsoffe
#define defsoffe     FClauses.First
#define dboffe       FRecords.First

typedef struct FUNCTOR {
    PTR    atoffe;      /* atom for this functor */
    union {
        unsigned Infoffe; /* general information field */
        struct {
            char Arityoffe; /* arity */
            char Flgsoffe; /* flags field */
        } FBytes;
    } FInfo;
    Header FClauses; /* clauses for this functor */
    Header FRecords; /* data base entries under
                     functor */
    PTR    nxtoffe; /* chain of functors with
                     same name */
    PTR    gtoffe; /* start of general skeleton for
                     this term. points to this entry. */
} FUNCTOR;
```

FIGURE 3-10 - FUNCTOR STRUCTURE DECLARATION

FIGURE 3-11 - FUNCTOR STRUCTURE DIAGRAM

Following is a brief description of the internal objects.

atoffe: atom of FUNCTOR entry - Pointer to the ATOM whose string is the principle functor.

FInfo: FUNCTOR Information - this union causes later entries in the structure to match positionally equivalent entries in the ATOM structure.

FBytes: FUNCTOR Bytes

Aritvoffe: Arity of FUNCTOR entry - number of arguments of of this FUNTOR.

Flgsoffe: Flags of FUNCTOR entry - use is identical to that for ATOMs.

FClauses: FUNCTOR Clauses/Records

FRecords: The FClauses.First and FClauses.Last entries point to

opposite ends of a doubly linked chain of CLAUSEREC structures (described next). FRecords pointers point to a similar chain of user defined records which are not evaluated in the normal course of unification. They must be accessed using the Internal Database commands described in section 5.10 of the C-Prolog users manual.

nxtoffe: next of FUNCTOR entry - pointer to the next functor entry in the chain of entries having the same principle functor but differing arities.

gtoffe: general term of FUNCTOR entry - pointer to this FUNCTOR. Note: the gtoffe entry serves in another structure which is always created immediately after the FUNCTOR. That is, this entry is also the Fn entry in a SKEL structure (described later).

3.2.4 The CLAUSEREC

Figure 3-12 is a reproduction of the declaration of the typedef of the CLAUSEREC. Figure 3-13 is a diagrammatic representation of the structure. This structure, together with SKEL's (described next) make up an instance or a "fact". Pay particular attention to the descriptions of hdofcl, bdyofcl, and altofcl.

ltofc1: (quantity of) local and temporary (variables) of clause

lvofcl: (quantity of) local variables of clause

gvofcl: (quantity of) global variables of clause

```

typedef struct CLAUSEREC {
    unsigned char ltofcl; /* no. of local and
                          , temp. variables */
    unsigned char lvofcl; /* no. of local vars */
    unsigned char gvofcl; /* no. of global vars */
    char infofcl; /* general inf. */
    int refcofcl; /* reference count */
    PTR hdofofcl; /* clause head */
    PTR bdyofcl; /* body of clause */
    PTR altofcl; /* alternatives */
    PTR prevofcl; /* previous clause */
} CLAUSEREC;

```

FIGURE 3-12 - CLAUSEREC STRUCTURE DECLARATION

FIGURE 3-13 - CLAUSEREC STRUCTURE DIAGRAM

ltofcl: (quantity of) local and temporary (variables) of clause

lvofcl: (quantity of) local variables of clause

gvofcl: (quantity of) global variables of clause

NOTE: The differentiation of variables as "local", "temporary", or "global" is conceptually different from similar descriptions in procedure oriented languages.

A global variable is one which occurs as an argument of a compound term which itself occurs as an argument in a

compound term. In the earlier example "supplies(bar, gadget(X)).", "X" is a global variable.

A local variable is one which occurs in more than one goal in a rule. In the rule "supplies(acme,X) :- widget(X).", "X" is local variable. What this really means is that the value of a local variable must be saved after a goal is solved, since it will be used in some later goal.

A temporary variable is one which does not need to be saved. In essence it is simply a place holder. Suppose we had a rule that stated that "acme" supplied any widget which was a subpart. In prolog, this would be "supplies(acme,X) :- subpart(X,Y), widget(X)." In this case "Y" is a temporary variable.

infofcl: information of clause - currently the 1-bit and 2-bit are used to determine whether a clause is erased and/or in use.

The 4-bit is used in Prolog to flag an "ingres.fact"

refcofcl: reference count of clause

hdofcl: head of clause - pointer to a SKEL structure which contains a representation of the head of this instance of this functor.

bdyofcl: body of clause - multi-use pointer. If NULL, this clause is a ground unit clause. If bdyofcl is an integer between 1 and 255, this clause is a built-in predicate and this value is the case label in a switch statement controlling execution of code associated with built-in predicates. If

it is anything other than the above then this is a pointer to a SKEL structure representing the body of this non-unit clause.

altofcl: alternate of clause - if non-null, there is another instance of this clause which may be tried if unification fails on this instance. ie. a pointer to the next CLAUSEREC in the chain.

prevofcl: previous of clause - CLAUSEREC's form a doubly linked chain. This is necessary because a new instance of a clause can be asserted at either the beginning of the chain (asserta(pred)) or at the end (assertz(pred)). This pointer, therefore, points to the previous CLAUSEREC in the chain.

3.2.5 The SKEL

Used extensively in the code, the SKEL structure is rather less structured than the declaration (shown in Figure 3-14) would lead one to believe. Figure 3-15 is a diagrammatic representation of the SKEL

```
typedef struct SKEL {
    PTR Fn;
    PTR Arg1;
    PTR Arg2;
} SKEL;
```

FIGURE 3-14 - SKEL STRUCTURE DECLARATION

FIGURE 3-15 - SKEL STRUCTURE DIAGRAM

structure. Progress substitutes a pointer to a new structure for the pointer normally stored in Fn.

In actual usage, up to 255 arg's may appear after the Fn entry. The arg's are addressed by first pointing at the structure and then stepping through the arg's.

Fn FUNCTOR - address of a FUNCTOR entry.

Arg1 Argument - multi-use objects, the Args may

Arg2 contain any of the following:

- global or local variable identifier.
- a constructed number (Float or Integer).
- a pointer to an ATOM, FUNCTOR or another SKEL.

3.2.6 The FRAME

A new FRAME is created for every goal encountered. Figure 3-16 is a reproduction of the declaration of the FRAME structure and associated definitions. Figure 3-17 is a diagrammatic representation of the structure.

The following explanations use the defined rather than the declared names.

```

    /* local frame */

typedef struct {
    PTR  Lcpofcf;    /* preious choice point */
    PTR  Gofcf;      /* goal */
    PTR  Gfofcf;     /* goal's local frame */
    PTR  Gsofcf;     /* global frame associated
                     with this frame */
    PTR  Trofcf;     /* tr at entry */
    unsigned Infofcf; /* general information */
    PTR  Cofcf;      /* continuation on entry */
    PTR  Altofcf;    /* alternatives */
} FRAMECTL;

/* size in cells */

#define szofcf    sizeof(FRAMECTL)/sizeof(PTR)
#define MAX_FRAME szofcf+MAX_VAR+1

typedef struct FRAME {
    FRAMECTL FCTL;
    PTR  v1ofcf; /* variables */
    PTR  v2ofcf;
    PTR  v3ofcf;
    PTR  v4ofcf;
    PTR  v5ofcf;
} FRAME;

#define lcpofcf  FCTL.Lcpofcf
#define gofcf    FCTL.Gofcf
#define gfofcf   FCTL.Gfofcf
#define gsofcf   FCTL.Gsofcf
#define trofcf   FCTL.Trofcf
#define infofcf  FCTL.Infofcf
#define cofcf    FCTL.Cofcf
#define altofcf  FCTL.Altofcf

```

FIGURE 3-16 - FRAME STRUCTURE DECLARATION

FIGURE 3-17 - FRAME STRUCTURE DIAGRAM

lcpofcf: last choice point of control frame - a pointer to the most recent previous frame which still has alternate instances of its associated FUNCTOR/goal.

gofcf: goal of control frame - a pointer into the instance of some parent goal which caused this goal to be invoked.

gfocf: goal frame of control frame - pointer to the FRAME of the parent goal.

gsofcf: globals of control frame - pointer into a global memory area where global variables of this goal may be stored if needed.

trofcf: trail of control frame - pointer to available area in trail memory. Information will be added to the trail stack if:

1. this instance succeeds and was not in use,

by some previous goal, and/or

2. A variable becomes instantiated as a result of the success of this goal.

infofcf: information of control frame - used primarily to store information concerning whether this goal can be traced.

cofcf: continuation of control frame - points to the next goal in the parent goal which is to be executed if this goal succeeds.

altofcf: alternate of control frame - address of the altofcl in the current CLAUSEREC. ie where to look for an alternate instance if this instance fails or some later goal fails requiring a new instance of this goal.

vlofcf: variable 1 of control frame - C-Prolog allows up to 255 arguments in a term and space for variable arguments is allocated here in the frame as necessary. The first five locations are named for convenience in dealing with built-in predicates. User defined arguments are referenced by their position relative to the frame.

/* molecule layout - the Sk field MUST be the first! */

```
typedef struct MOL {
    PTR Sk;
    PTR Env;
} MOL;
```

FIGURE 3-18 - MOL STRUCTURE DECLARATION

FIGURE 3-19 - MOL STRUCTURE DIAGRAM

3.2.7 The MOL

The MOLecule is used to control execution of user's goals. A reproduction of its declaration is shown in figure 3-18 and its diagrammatic representation is shown in figure 3-19.

Sk Skeleton - contains a pointer to the gtoffe of a FUNCTOR or a pointer to an ATOM.

Env Environment - a pointer to the start of a list of chains of pointers (possibly singular) associated with the arguments of the FUNCTOR pointed to by Sk.

3.3 MEMORY ALLOCATION

Memory in C-Prolog is managed in six contiguous areas. The sizes of these areas can be initialized during C-Prolog start-up, but cannot be changed dynamically. Figure 3-20 shows their relative locations. Note: Only those areas which have the word "stack" in brackets after the name, function in the true sense of a stack. The Heap has a garbage collection function associated with it, so its usage does increase and decrease dynamically but not in the same sense as a stack.

atom0:	Atoms
auxstk0:	Auxiliary (stack)
trbase:	Trail (stack)
skel0:	Heap (input terms)
glb0:	Global (stack)
lcl0:	Local (stack)

FIGURE 3-20 - PROLOG MEMORY LAYOUT

```

lookup("atom")
Begin
    ptr = hash_table_loc("atom");
    while (*ptr)
        Begin
            ptr = *ptr;
            if (((ATOM*)(ptr)->stofae) == "atom")
                then return ptr
            else ptr = ((ATOM*)(ptr)->nxtofae)
        End
    *ptr = free_atom_area_ptr;
    ptr = *ptr;
    Bump(free_atom_area_ptr) by
        sizeof(ATOM) + stringlength("atom");
    Build_atom_at_ptr("atom");
    return ptr;

```

FIGURE 3-21 - ATOM LOOKUP AND INSTALLATION FUNCTION

The Atom work area contains a 128 word hash table and ATOM structures. Whenever an atom is encountered (either as a predicate

name or as an argument) it is entered as an ATOM structure in this area. The pseudo code in figure 3-21 shows how this is accomplished.

As currently implemented, there is no way to determine whether an ATOM is currently pointed to by a FUNCTOR, an argument in some instance, or by nothing at all. There is, therefore, no way to remove unwanted or unused ATOM's and usage of the Atom work area never decreases during a C-Prolog session. This has serious implications which will be discussed in chapter VI.

The auxiliary stack is used to store information about variables which have been read in, and retains this information only so long as it is required to complete the operations associated with the current goal.

The trail stack is used to track the first occasion when a goal is invoked. This is done so that, if a later goal causes the instance to be erased, its actual erasure will not take place until all goals, where it was in use, have been satisfied. The trail also stores pointers to variables when they are instantiated so that if a later goal fails, variables will be reset at appropriate times.

The FUNCTOR, CLAUSEREC, and SKEL structures are all stored in the Heap.

MOLEcules are the only declared structures stored in the global stack, while FRAMEs are the only declared structures appearing in the local stack. Both stacks however, are used for temporary storage of arbitrary values such as strings, lists, and PTRs as necessary.

3.4 Conclusion

The object 'PTR' is used to address virtually all structures in C-Prolog. Therefore, a structure's type is most often determined by its relative memory location rather than by its declared type. This makes the C-Prolog code very adaptable but also very difficult to follow.

The next chapter will show how the interpreter's unification algorithm uses the structures described here and linkages between them, to attempt to prove the truth of a user's goal. That is to find answers to the user's query.

Chapter Four

PROLOG INTERPRETER OPERATION

4.1 Introduction

The C-Prolog interpreter is a practical implementation of a recursive depth first search and pattern matching algorithm. The idea is to find a fact which either directly matches an input goal or which matches subgoals of a rule whose head matches the goal.

This chapter describes the algorithm of the C-Prolog interpreter. First an overview is obtained by describing in very broad terms how the interpreter solves a user's goal. Then, a more detailed examination of the flow of control mechanisms is carried out. Several examples are used in conjunction with pseudo code to lead the reader through the complete operation.

4.2 General Operation

To show how the interpreter works, let us examine how C-Prolog goes about solving the example of chapter I. ie. "widget(Z), print(Z), nl, fail."

First the database of figure 1-4 must be read into memory. This causes ATOMs to be built for each of the strings and FUNCTORs to be built for each of supplies/2, subpart/2, gadget/1, and widget/1. CLAUSEREC's are chained together and pointers to appropriate chains placed in the FUNCTOR's. In each CLAUSEREC will be a pointer to its SKEL structure which represents the "head" of the clause. In the glossary it is noted that a ground unit clause ("supplies(foo,pl)" for

instance) is simply a clause without a "body". In those CLAUSEREC's associated with the rules, there will also be a pointer to a SKEL representing the "body" of the clause. In each SKEL will be a pointer to its associated FUNCTOR and variable indicators, numbers or pointers to ATOMs or other SKELs. Review figures 3-4 to 3-7 to get this picture clearly.

When the goal is entered by the user, it is parsed and a binary tree structure containing MCLs and pointers is built in global memory. Storage space for "Z" is also reserved there. A pointer to the head of the tree is returned. Remember that the user's goal is a conjunction of goals. That is, "Find a widget 'Z' AND print it AND print a newline AND fail". This can be represented diagrammatically as shown in figure 4-1.

FIGURE 4-1 - DIAGRAMATIC REPRESENTATION OF USER'S GOAL

This structure is used by prolog to control the execution of the

user's goal. All the words in the structure are actually pointers to FUNCTORS. With the exception of "AND", these were located during parsing by first finding the appropriate ATOM and then following the functor chain pointers.

The PROLOG processes try to solve the goals by a depth first, left to right ordering which results in a search tree like the one shown in figure 1-5.

To solve the first AND, the interpreter first tries to find a solution to the goal widget(Z). To do this, it uses the pointer to widget/1 to locate the first instance of widget/1 (i.e. widget(w1)). Because "Z" is a variable, it successfully unifies with "w1" and a pointer to the "w1" ATOM is placed in the storage location in global memory reserved for "Z". Now the right side of the first "AND" must be solved. The right side is another "AND". To solve this "AND", the print(Z) goal must be solved. Print/1 is a built-in predicate so the entry "FClauses->First" in its FUNCTOR will contain a case label which causes code to be executed which prints "w1" to the standard output. This solves the print(Z) goal so the right side of the second "AND" is attempted. Again, this is an "AND" whose left side is solved in a similar fashion to the print/1 goal and a newline is output to the screen successfully completing this goal. Now the "fail" goal is attempted. The code for this goal is absurdly simple. It simply causes a branch to the code normally executed whenever unification fails for any goal. This code searches for a previous goal which could be solved with a new clause.

During all of the above operations, a FRAME was built in local memory whenever a goal was encountered. Before an attempt to unify the arguments of the user goal with the arguments of the SKEL of the current instance is made, the "altofcl" (alternate of clause) entry is checked to determine if there is another instance in the chain of CLAUSERECs. If there is then this FRAME is pointed to by a variable called lcp (last choice point). If there is not then the lcp is not modified. In either case, the value of lcp is stored in the FRAME before the test so that there is always a linkage to a previous FRAME that had a choice point.

In the current case, none of print/1, nl, nor fail has an alternate, so the last choice point is the FRAME for "widget(Z)". Also stored in the FRAME for "widget(Z)", was the address of the alternate of clause (altofcl) pointer in the CLAUSEREC associated with widget(w1). This is used to find the CLAUSEREC associated with widget(w2) and pointers in the FRAME are adjusted. Now the pointer to the "w2" ATOM is put into the "Z" storage location and the whole routine for print(Z), nl, and fail is repeated. The same things occur for "w3". At this point, we have searched for and found successes 1, 2, and 3 in the search tree of figure 1-5.

In each of the above cases the bdyofcl (body of clause) pointer in the CLAUSEREC was examined and found to be NULL. At this point, however, the instance associated with the widget rule is encountered.

After unifying "X" and "Z" (pointers are simply adjusted) the algorithm attempts to solve the "body" goals. Looking at figure 3-5,

it can be seen that bdyofcl points to a SKEL containing a pointer to the ,/2 ("AND") FUNCTOR and pointers to two other SKELs. The algorithm therefore, tries to solve the left branch of this tree. ie. it tries to solve "subpart(X,Y)". It uses the pointer in the SKEL to locate subpart/2 (figure 3-6) and then the first instance "subpart(w4,g1)". Remember, at this point neither "X" nor "Y" is instantiated but "X" points to "Z". Unification therefore, successfully takes place and the subpart goal succeeds causing "X"/"Z" to be instantiated to "w4" and "Y" to point to the "g1" ATOM. Now the right branch of the body of the rule must be solved. Therefore, the second SKEL in the body is examined and its first pointer (Fn) points to gadget/1 (figure 3-7). This is used to find the first instance of gadget/1 which turns out to be "g1", which matches the current instantiation of "Y", so the rule succeeds (success #4, figure 1-5) and the remainder of the user's goals are executed.

When the user's "fail" goal is executed, the last choice pointer is pointing to the FRAME which was built for gadget/1. Therefore, the remaining instances in figure 3-7 are all examined but none match on "g1". It fails and a new instance is obtained for subpart (subpart(w5,g2)). Now we move forward again to try and solve gadget(g2). The first instance in figure 3-7 (gadget(g1)) doesn't match but the second (gadget(g2)) does (success #5) so again the rest of the user's goals are executed resulting in a failure and the remaining gadget instances are again attempted. They fail to unify and subpart(w6,g2) is picked up which causes gadget to succeed on its

second attempt (success #6) and the user's goals are again executed.

Now whenever subpart finds a new instantiation for "Y", gadget will fail. Eventually subpart will fail causing the widget rule to fail. Since there are no more widget instances (see figure 3-5), the user's "widget(Y)" goal will fail causing the entire user's goal to fail.

The remainder of this chapter uses examples and pseudo code to describe, in much finer detail, the actual internal operation of the interpreter.

4.3 Flow of Control

To understand the flow of control of the C-Prolog interpreter, one must first realize that there are actually two levels of control. The first, written in prolog, might be called a meta-level, while the second, written in "C", is the interpreter's implementation of the unification algorithm. The two levels complement each other in a way that permits the "C" level to treat both built-in and user defined predicates in a similar fashion.

4.3.1 The Meta-Level Algorithm

Figure 4-2 is a code segment from the C-Prolog initialization file. It shows the first few levels of the system's session goal. When C-Prolog is executed, the the interpreter immediately attempts to solve the goal \$live/0. This attempt causes the goal \$cycle/1 to be tried. \$cycle/1 does some housekeeping (saving the current prompt) and then displays the symbol "| ?-" on the terminal and waits for an input. Note that the predicates \$prompt/1 and \$read/1 are "one of a

kind" (ie only one instance) and are never re-attempted on back tracking. This means that the \$repeat predicate (which always succeeds) will cause the prompt to be redisplayed and \$read/1 to be reexecuted whenever any subgoal fails. In this case, the only subgoal which can fail is \$interpret/2. Note also that "behind the scenes" in the \$read predicate, the entire process of parsing and creation of data structures described in chapter III takes place.

Examining the \$interpret/2 predicate, one can see that the only case where it can succeed is if the value returned by \$read/1 is "end-of-file". If this value is encountered, \$cycle/1 will succeed and the predicate \$rest_in_peace/0 will be executed which causes all files to be closed except standard input/output and then writes the sign-off statement and exits. If the user enters the built-in predicate halt/0, the same code as for \$rest_in_peace/0 is executed.

```
$live:- $cycle(0), $rest_in_peace.  
      % this is the Prolog session goal
```

```
$cycle(0):-  
    prompt(Old, '| '),  
    $repeat,  
        $prompt('| ?- '), $read(C),  
        $interpret(C,0),  
    prompt(_,Old),  
    !.
```

```
$cycle(Status):-  
    prompt(Old, '| '),  
    $repeat,  
        $read(C),  
        $interpret(C,Status),  
    prompt(_,Old),  
    !.
```

```
$interpret(C,Status) :-  
    var(C),  
    !,  
    display('! Statement is a variable'),  
    ttylnl,  
    fail.
```

```
$interpret(end_of_file,_):-!.
```

```
$interpret(C,Status):-  
    $directive(C,Status,D,T),  
    !,  
    prompt(Old, '|: '),  
    $dgoal(T,D),  
    prompt(_,Old),  
    fail.
```

FIGURE 4-2 - INITIALIZATION FILE
(continued next page)

```

$interpret(C,Status):-
    expand_term(C,C1),
    $assert(C1),
    !,
    fail.

$assert(C):-$assertr(C),!.
$assert(C):-
    seeing(user),
    !.
$assert(C):-
    display('! clause: '),
    display(C).

$directive(:-(X),_,X,command) :- !.
$directive(?-(X),_,X,question) :- !.
$directive(X,0,X,question).

$dogoal(command,C) :-
    $user_exec(C),
    !.

$dogoal(command,_) :-
    !,
    ttynl, display('?'),
    ttynl.

$dogoal(question,Q) :-
    $save_read_vars,
    $user_exec(Q),
    $reply,
    ttynl, display(yes),
    $reset_read_vars,
    ttynl,
    !.

$dogoal(question,_) :-
    ttynl, display(no),
    $reset_read_vars,
    ttynl,
    !.

```

FIGURE 4-1 - INITIALIZATION FILE

Further examination of the \$interpret/2 predicates reveals that

the first is simply an error check to make sure that the user doesn't try to ~~execute~~ a variable as a predicate. The third and fourth \$interpret/2 predicates are where the actual interpretation takes place.

Before examining the subgoals of \$interpret/2, the variable "status" should be briefly explained. If we are at the top level of the interpreter, that is, all we are doing is entering goals for the interpreter to solve, then "status" will be 0. However, if we have caused the interpreter to consult or reconsult a file, then a new level of the \$cycle/1 predicate with its argument not equal to 0 will have been invoked and therefore "status" will have some non-zero value. Recall that, in a file being consulted, if we want a command executed or a goal solved, we must preface the predicate with either ":-" or "?-". Any other entry in the file is something to be asserted or reasserted.

Returning now to the subgoals of the third \$interpret/2 predicate, we see that for the \$directive/4 goal to succeed, either status must be zero or the term read in must start with either ":-" or "?-". If \$directive/4 succeeds then, after taking care of housekeeping requirements, the predicate \$dgoal/2 is executed.

The difference between prompt/2 and \$prompt/1 should be noted here. Prompt/2 simply causes the string in the second argument to be saved in a "C" variable called "atprompt" while the current atprompt value is moved into the first argument. The execution of \$prompt/1, on the other hand, causes its argument to be written to the standard

output if the current file is standard input.

The predicate `$dogoal/2` attempts to do just what its name implies. The third `$dogoal/2` predicate is the one of primary interest to us. When invoked, this goal first adjusts pointers in the auxiliary stack (`$save_read_vars/0`) to ensure that any variables in the input goal are protected. Then it executes `$user_exec/1` which simply does a kind of sleight of hand to cause the interpreter to execute the argument of `$user_exec/1` as a predicate requiring unification. If this is successful, `$reply/0` causes the instantiated variable (if any) to be displayed and then checks input to determine if the user is satisfied with the response or wishes unification to try again. If the response is `;<cr>`, ie, try again, `$reply/0` simulates a failure causing the unification algorithm to try for a new instantiation of the variables of the user goal. If the user was satisfied or there were no variables in the user goal, then `$reply/0` succeeds and `$dogoal/2` will succeed. If the third `$dogoal/2` fails either because there could be no unification on the user goal or because the user was never satisfied with the response then the fourth `$dogoal/2` predicate is executed which cannot fail.

Now, return to the third `$interpret/2` predicate. Since `$dogoal/2` cannot fail once invoked, `prompt/2` will swap the old prompt back into `atprompt` and `$interpret/2` will fail. Because there was a cut after `$directive/4` and all the `$dogoal/2` predicates have "cut's" in them the failure causes `$interpret/2` to fail completely. This results in a "fail back" to the `$repeat/0` goal in `$cycle/1`, a new prompt to be

displayed and \$read/1 to await a new input.

If \$directive/4 in the third \$interpret/2 fails, then we know that we are attempting to assert a term and, after expanding the term, this is done. Again, as in \$dogoal/2, \$assert/1 always succeeds (from a prolog viewpoint) and therefore, \$interpret/2 will fail causing the \$repeat/0 in \$cycle/1 to be invoked again.

Any greater study of the meta-level interpreter is not necessary to an understanding of the operation of C-Prolog. However, Appendix A contains the complete initialization file.

4.3.2 The "C" Algorithm

Figures 4-3 through 4-14 contain a commented pseudo code version of the C-Prolog search and unification algorithm. The variables are as self explanatory as possible and type-casting is only shown where absolutely necessary to clarify an operation. Some housekeeping operations have been left out or simply commented upon. Certain "if then else" decisions that in the actual code are based on relative addressing have been modified to more clearly show what the decision is based on. The following examples are intended to serve as guide to understanding the algorithm. Before working through the examples, the reader should review the introduction to chapter III to be sure he understands the broad outlines of the prolog control process.

```

PTR pg, /* parent goal pointer - a pointer to
        either a MOL or a SKEL */
goal, /* current goal - a pointer to a SKEL */
lcp, /* last choice point - points to a FRAME */
pgf, /* parent goal frame pointer - points to a
      FRAME */
glbf, /* current global frame pointer - a global
      frame is normally a MOL with associated
      pointers */
pglbf, /* pointer to global frame associated with
      parent goal */
cgf, /* pointer to FRAME associated with goal */
cont, /* pointer to next SKEL if current goal
      succeeds */
pip1, /* intermediate parent pointers - used to */
pip2, /* keep track of chains of argument pointers */
pip3, /* in parent goal */
gip1, /* current goal intermediate pointers */
gip2,
gip3,
faillabel, /* pointer to altofcl in current
            CLAUSEREC */
clauseptr, /* pointer to current CLAUSEREC */
functorptr; /* pointer to current FUNCTOR */
/* This pseudo-code accurately reproduces the
control flow concepts of the C-Prolog interpreter.
However, the original code is all "in-line" and
therefore difficult to follow. To alleviate this, the
pseudo-code introduces a mainline loop which makes
procedure calls based on the value of an integer set by
preceding procedures. */
int switchvalue, /* mainline caselabel */
caselabel, /* evaluable predicate caselabel */
arity, /* current FUNCTOR's number of args */
define interpret_goal 1;
define alternate_instance 2;
define body_check 3;
define resolve_args 4;
define eval_pred 5;
define body_exists 6;
define continuation 7;
define no_body 8;
define ifail 9;
define gfail 10;

```

FIGURE 4-3 - Pseudo-Code Declarations

```
1. While (TRUE)
2.   Begin
3.     switch (switchvalue)
4.       Begin
5.         interpret_goal: switchvalue = ical1(); break;
6.         alternate_instance: switchvalue = alt(); break;
7.         body_check: switchvalue = bcheck(); break;
8.         resolve_args: switchvalue = rargs(); break;
9.         eval_pred: switchvalue = epred(); break;
10.        body_exists: switchvalue = ebody(); break;
11.        continuation: switchvalue = carry_on(); break;
12.        no_body: switchvalue = neck_foot(); break;
13.        ifail: switchvalue = instance_failed(); break;
14.        gfail: switchvalue = goal_failed(); break;
15.      End /* of switch */
16. End /* of loop */
```

FIGURE 4-4 - Main Control Loop of Interpreter

```

1. icall()
2. Begin
3.   if (pg points to a SKEL) /* actually determined*/
                               /* by relative addressing*/
4.   then goal = pg; /* this is the address of an ATOM
                       /* or the address of a gtoffe of a FUNCTOR */
5.   else Begin
6.     pglbf = ((MOL*)(pg))->Env;
7.     goal = ((MOL*)(pg))->Sk;
8.   End
9.   functorptr = *goal;
10.  clauseptr = functorptr->FClauses.First;
    /* Because FUNCTOR's and ATOM's are identically
    structured in the first 7 words, they can be treated
    identically for the objects in those 7 words. In
    addition, it doesn't matter whether goal contains the
    address of a gtoffe or the address of an ATOM because
    the contents at that address will be the address of the
    appropriate structure. If the structure is an ATOM
    however, clauseptr may now contain an integer
    representing the case label of an atomic built-in
    predicate such as !(cut) or abort. */
11.  if (0 < clauseptr < 255)
12.  then Begin
13.    case_label = clauseptr; return eval_pred;
14.  End
    /* CLAUSEREC's which have been retract'd are not
    immediately unlinked. Instead, a bit in the CLAUSEREC
    is set and a garbage collection algorithm later cleans
    up the Heap */
16.  while (clauseptr && erased(clauseptr))
17.    clauseptr = clauseptr->alstofcl;
18.  if (!(clauseptr)) return gfail;
    /* build a frame */
19.  cgf->lcpofcf = lcp;
20.  cgf->gofcf = pg;
21.  cgf->gfofcl = pgf;
22.  cgf->gsqfcl = glbf;
23.  lcp = cgf;
24.  return alternate_instance;
25. End

```

FIGURE 4-5 - Predicate Resolution

```

1. alt()
2. Begin
3.   faillabel = &(clauseptr->altofcl);
4.   if (!(*faillabel)) lcp = cgf->altofcl;
5.   clear space after current FRAME for local and
6.   temporary variables
7.   do the same for global variables in global
8.   cgf->altofcl = faillable;
   /* at this point the predicate name has been matched
   if there are no arguments, check for a body */
9.   if (IsAtomic(goal)) return body_check;
10.  return resolve_args;
11. End

```

FIGURE 4-6 - Alternate Instance Handling

```

1. bcheck()
2. Begin
3.   if (!(clauseptr->bdyofcl))
4.     then Begin
5.       pg = cgf->gofcf;
6.       cgf = cgf + (sizeof(FRAME) +
                      # of local variables);
7.       return no_body;
8.     End
9.   else Begin
10.    cont = clauseptr->bdyofcl;
11.    pgf = cgf;
12.    pglbf = glbf;
13.    cgf = cgf + (sizeof(FRAME) +
                  # of local variables);
14.    if (0 < cont < 255)
15.      then Begin
16.        caselabel = cont;
17.        return eval_pred;
18.      End
19.    End
20.  return body_exists;

```

FIGURE 4-7 - Body Goals Handling

```

1.  rargs()
2.  Begin
3.      gip1 = clauseptr->hdofcl;
4.      pip1 = goal;
5.      arity = ((SKEL*)(gip1))->Fn->arityoffe;
6.      while (arity-- > 0)
7.          Begin
8.              gip2 = *++gip1;
9.              if (VariableIndicator(gip2))
10.                 then Begin
11.                     Use gip2 to get proper offset into
12.                     local or global frame;
13.                     Use gip3 and gip2 to bootstrap to the
14.                     actual instantiation (if any) leaving
15.                     gip2 containing the value and gip3
16.                     pointing at the values address;
17.                     pip2 = *++pip1;
18.                     resolve pip2 and pip3 as for gip2, gip3
19.                     compare pip2 and gip2;
20.                     Both are variables? *gip2 = pip3;
21.                                     continue loop;
22.                     gip2 only is a variable? *gip2 = pip2;
23.                                     continue loop;
24.                     pip2 only is a variable? *pip2 = gip2;
25.                                     continue loop;
26.                     Both point at FUNCTORS? call a new
27.                     unification algorithm
28.                     Success? continue loop
29.                         else return ifail;
30.                     Both point at ATOM's or Primitives?
31.                     gip2 == pip2? continue loop
32.                         else return ifail;
33.                     None of the above? return ifail;
34.                 End
35.             else Begin
36.                 pip2 = *++pip1;
37.                 resolve pip2, pip3 as above;
38.                 compare pip2, gip2 and choose as above;
39.             End
40.         End /* loop */
41.     return body_check;
42. End

```

FIGURE 4-8 - Argument Resolution

```

1. epred()
2. Begin
3.     switch (case_label)
4.     Begin
5.         1: /* code for AND */
6.         2: /* code for HIDDEN_CALL*/
7.
           170: /* the highest current case */
/* generally the various cases return
"continuation" if successful or "gfail" if
not. Some of course return to other
procedures and some like exit never return*/
End

```

FIGURE 4-9 - Built-in Predicate Evaluation

```

1. e_body()
2. Begin
3.     if (*cont points to ,/2 FUNCTOR) /* ie the next
                                           /* goal is not the
                                           /* last in the body */
4.     then Begin
5.         pg = ((SKEL*)(cont))->Arg1;
6.         cont = ((SKEL)(cont))->Arg2;
7.     End
8.     else Begin
9.         pg = cont;
10.        cont = NULL;
11.    End
12.    return interpret_goal;
13. End

```

FIGURE 4-10 - Goal Tracking

```

1. carry_on()
2. Begin
3.   if (cont) return body_exists;
4.   if (pgf > lcp) cgf = pgf; /* ie if the goal
                               /* which just succeeded
                               /* doesn't have an
                               /* alternate instance
                               /* don't save the frame
                               /* it is not
                               /* re-useable*/
5.   cont = pgf->cofcf; /* we're at the foot of the
                       /* current goal so continue
                       /* with the body of the
                       /* parent goal */
6.   pg = pgf->gofcf;
7.   pgf = pgf->gfofcf;
8.   return no_body; /* there is no body of the
                     /* current goal, so see if we
                     /* got one from the parent
                     /* frame*/

```

FIGURE 4-11 - Continuation Handling

```

1. neck_foot()
2. Begin
   /* This procedure must be separate because entry
   /* can be as the result of solving a goal with
   /* no body or because the goal in the body has
   /* been solved */
3.   pglbf = pgf->gsofcf;
4.   return continuation;
5. End

```

FIGURE 4-12 - End-of-goal Handling

```
1. instance_failed()
2. Begin
3.   clear local and global variables; /* some may
   .                                     /* have been instantiated
   .                                     /* before the instance
   .                                     /* failed*/
4.   clauseptr = *faillabel;
5.   while ( clauseptr && erased(clauseptr))
6.     clauseptr = clauseptr->altocf;
7.   if (clauseptr) return alternate_instance;
8.   lcp = cgf->lcpofcf; /* this goal failed
   .                                     /* completely so find the last
   .                                     /* choice point and cause a
   .                                     /* fail-back to the last
   .                                     /* parent having a choice
   .                                     /* point */
9.   return gfail;
10. End
```

FIGURE 4-13 - Instance Failure Handling

```

1. goal_failed()
2. Begin
3.   while (TRUE) /* find a choice point FRAME having
                     /* a CLAUSEREC that is not erased */
4.     Begin
5.       for (clauseptr = *(lcp->altofcl);
6.           clauseptr && erased(clauseptr);
7.           clauseptr = clauseptr->altofcl);
8.       if (clauseptr) break;
9.       lcp = lcp->lcpofcf;
10.    End
11.  clear local and global variables;
    /* reset pointers */
12.  cgf = lcp;
13.  pgf = cgf->gfocf;
14.  pg = cgf->gofcf;
15.  cont = cgf->cofcf;
16.  if (pg points to a SKEL)
17.  then Begin
18.    goal = pg;
19.    pglbf = pgf->gsofcf;
20.  End
21.  else Begin
22.    goal = ((MOL*)(pg))->Sk;
23.    pglbf = ((MOL*)(pg))->Env;
24.  End
25.  return alternate_instance;
26. End

```

FIGURE 4-14 - Goal Failure Handling

As the first example, suppose we have consulted the file represented by figure 1-4 and therefore have, in memory, the structures represented by the memory pictures of figures 3-4 to 3-7.

Let the following prolog query be entered:

```
subpart(w5,g2). /* Is w5 a subpart of g2? */
```

During the \$read/1 operation, this will be parsed and in global memory will be a MOL structure and ancilliary pointers which can be thought of as looking like figure 4-15.

FIGURE 4-15 - Global Frame for "subpart(w5,g2)"

A pointer to the MOL will be stored in vlofcf of the Sread/1 FRAME. During the evaluation of Sinterpret/2, Sdirective/4, Sdogoal/2 and eventually Suser_exec/1, this pointer will be handed off from FRAME to FRAME. When the code for the Suser_exec/1 label of the switch statement in epred (figure 4-9) is executed, this pointer is moved into "cont" and "continuation" is returned to the main loop.

The main-loop switch now causes the code in figure 4-11 to be executed. At line 3 cont is not NULL so body_exists is returned and main will cause e_body (figure 4-10) to be executed. The contents of "cont" is a pointer to the subpart/2 FUNCTOR so pg is set equal to cont, cont is NULL'd and interpret_goal is returned. This causes icall (figure 4-5) to be called.

In icall(), pg is not a pointer to a SKEL, so pg1bf is set equal to pointer "a" (figure 4-15) and goal equals a pointer to the gtoffe of subpart/2. Functorptr is set to point to the subpart/2 FUNCTOR. Clauseptr picks up the value of "functorptr->EClauses->First and,

since no clauses have been erased, a FRAME for subpart(w4,g1) is built. This FRAME will contain a pointer (altocf) to altocf1 (alternate of clause) in the subpart(w4,g1) CLAUSEREC so that if this fails to unify (it will), the next instance (subpart(w5,g2)) can be found. Note that "cont" was NULL'd at line 10 in figure 4-10, so cgf->cofcf will be NULL. Line 24 is executed so alternate_instance is returned and alt() (figure 4-6) is called. In alt() note that (line 3) faillabel is set to the address of clausptr->altocf1 and not the contents. In our current example the contents of faillabel is a pointer to the subpart(w5,g2) CLAUSEREC (ie, not NULL), so the last choice point is this FRAME. We have two local "variables" here, so two words above the current frame (actually vlofcf and v2ofcf) are cleared. There are no global variables.

Since goal is not atomic, line 10 is executed and main calls rargs() (figure 4-8). (This is the pseudo code of the actual unification algorithm.) The resolution process involves using intermediate pointers for both the parent goal and the goal. Gip2 is pointed to the "w4" ATOM in line 8 so line 26 is branched to. The SKEL arguments after the gtoffe always indicate a global variable. Pglbf is used as a start point to chain through pointers looking for either a NULL (an uninstantiated variable) or a pointer to a primitive, ATOM, or FUNCTOR. In this case pip2 will point to the "w5" ATOM and, when compared to gip2 which points to the "w4" ATOM, the resolution will fail causing ifail to be returned.

Main now calls instance_failed() (figure 4-13). Since this is a

"current goal" failure, instead of one where we have failed on a later goal and are re-trying this goal, faillabel is used directly instead of `cgf->altofcf(alternate of control frame)` (line 4). Line 4 sets `clauseptr` pointing to the CLAUSEREC for (w5,g2) and since none of the clauses have been erased we return `alternate_instance` at line 7 causing main to call `alt()` (figure 4-6):

At line 3, `faillabel` is updated so that if this CLAUSEREC fails to unify, the next CLAUSEREC will be picked up. There is another CLAUSEREC, so this FRAME is still the last choice point (line 4). At line 5, we again NULL the variables just in case any were instantiated on the previous attempt to resolve the arguments. We update `cgf->altofcf` (point it at `altofcl` in the CLAUSEREC of (w5,g2)) at line 8 in case a later goal causes a "fail back" to this goal, and "return `resolve_args`" (line 10) is executed so that main calls `rargs()` (figure 4-8).

This time, `gip2` and `pip2` both point at the "w5" ATOM so the loop gets traversed a second time and both `gip2` and `pip2` will point to the "g2" ATOM which means that the loop can be exited and line 31 gets executed causing `bcheck()` (figure 4-7) to be called. At line 3 we find that there is no body of clause for "goal", so `pg` is set equal to the goal that caused this FRAME (in this case the MOL of fig 4-15). "subpart(w5,g2)" has no local variables (only temporaries). `Cgf` will point to what is currently `cgf->vlofcl` (line 6) and return `no_body` (line 7) is executed. This is done to protect the subpart/2 FRAME since there are still alternate CLAUSERECs for subpart.

Neck_foot() (figure 4-12) is called. This procedure is really a start point for working back up the tree of goals to find a parent which still has more body goals to be proven. As a result, we immediately return "continuation" resulting in a call on carry_on() (figure 4-11). "cont" was NULL'd in e_body. Lines 4 through 8 get executed and no_body is returned. Note that pg and pgf are now associated with \$dogoal/2 and cont points to a ,/2 SKEL between \$repply/0 and the rest of the body of \$dogoal/2. Note also that cgf still points to a vacant area above the FRAME for subpart/2. The subpart/2 FRAME is protected in this manner because there are still choices possible, if required, for subpart/2.

Neck_foot() (figure 4-12) is again called to cause the parent goal to be stepped back and then to cause carry_on() (figure 4-11) to again be executed. This time cont is not NULL and body_exists is returned. The body we are now talking about is the rest of the goals of \$dogoal/2. E_body() (figure 4-10) is called and, because cont now points to the ,/2 SKEL between \$repply/0 and the rest of the body, pg is set to the contents of the first arg of the ,/2 SKEL and cont is set to the contents of the second arg. In this case there are no variables in the original goal, so \$repply/0 simply returns continuation and the remainder of the \$dogoal/2 goals are executed. This results in "yes" being printed and then the normal \$interpret/2 failure which in turn causes the prompt to be displayed and \$read/1 to wait for input.

Suppose the modified query, "subpart(X,g2)" (what are the

subparts of "g2"?), were entered. Before continuing with this example, it should be noted that in this case a structure containing information about "X" will be stored in the auxiliary stack: \$save_read_vars/0 does nothing more than protect this structure.

FIGURE 4-16 - Global Frame for "subpart(X,g2)"

Now, after the \$read/1 has executed, global memory will contain the MOL and pointers shown in figure 4-16. The resolution process will cause the NULL to be replaced by a pointer to the "w5" ATOM. Now, when the code for \$reply/0 is executed, "X = w5" is displayed and the terminal input is read to determine if the result is satisfactory.

Suppose the user enters ";<cr>". This causes the \$reply/0 code to return gfail resulting in a call on goal_failed() (figure 4-14). Since \$reply/0 does not have an alternate, lcp still points to the FRAME for subpart/2 and, therefore, clauseptr will immediately get the address of the CLAUSEREC for subpart(w6,g2) (lines 4-10). Note: Clearing the global variables (line 11) causes what is now a pointer to the "w5" ATOM to be returned to the NULL state of figure 4-16.

22

OF / DE

After setting `cgf` to `lcp` (ie pointing at the `subpart/2` FRAME) all other necessary pointers can be returned to their condition as though this was a normal entry to this instance (lines 12 - 24) and `alternate_instance` (line 25) is returned.

Resolution will now cause the NULL in figure 4-16 to be replaced with a pointer to the "w6" ATOM and `$reply/0` will display "X = w6". If we again enter "; <cr>" essentially the same things will happen except that each time a new clause is selected, it will not be able to unify on `arg2` ("g2") and return ifail will cause `instance_fail` to execute until eventually `clauseptr` will test NULL at line 7 (figure 4-13). At this point the goal `subpart(X,g2)` has failed completely and we return `gfail` which, in effect, causes `$user_exec/1` to fail. Using `lcp` and `lcp->lcpofcf` we will eventually fail back to the FRAME for `$dgoal/2` since it is the next previous goal which had a choice point.

Now the last `$dgoal/2` CLAUSEREC is found and it cannot fail since all that it does is print "<cr> no <cr>". Thus `$dgoal/2` will succeed and `$interpret/2` will fail causing the `$repeat/0` in `$cycle/1` to be executed, the prompt to be displayed and `$read/1` to be executed.

4.4 Conclusion

An instance (CLAUSEREC plus associated SKELs) is the core object in C-Prolog. In essence the instance actually describes a particular clause.

The equivalent object to an instance in a relational database is the tuple where the relation name is equivalent to the predicate name

and the attributes are equivalent to instantiated arguments of a ground unit clause.

The object of interfacing C-Prolog to ingres is to permit C-Prolog to view a tuple as an instance. C-Prolog requires the ability to pick up a new instance of a clause if the current one fails or a later goal fails. It therefore has a pointer, `altocl`, in each instance which points to the alternate clause if there is one. An equivalent of this in ingres would be a pointer to next tuple. ie a pointer to the tuple which logically follows the one being currently retrieved.

As can be seen in the previous examples, C-Prolog depends extensively on the use of pointers. If a modification of the code could be made which would allow a pointer to a tuple to be functionally equivalent to a pointer to an instance, then, from the user's standpoint, the data in an ingres database becomes simply a large store of ground unit clauses. This pointer would substitute for `altocl` (alternate of clause) as long as there was a "next tuple" available.

The above describes the ideal interface between ingres and C-Prolog. The actual interface does not achieve the ideal, but it does provide essentially the same user's viewpoint.

Chapter V describes the actual implementation of the interface and demonstrates that it does provide the user with a system where the ingres database appears to be a large store of ground unit clauses.

Chapter Five

PROGRES DESCRIPTION

5.1 Introduction

The requirement for the prototype was basically to demonstrate that an interface could be built which would permit tuples of a relation to be used as prolog ground unit clauses. That is $\text{relation_name}(\text{tuple}) \equiv \text{predicate_name}(\text{arguments})$. As noted in chapters III and IV, the operation of prolog depends almost exclusively on pointers for resolving goals. In chapter IV we also noted that, if a pointer to a tuple could be made functionally equivalent to a pointer to an instance, we would have essentially the interface we wanted. To achieve this, we require that a pointer to the "next tuple" be returned by ingres with each retrieval. By "next tuple" we mean the tuple which logically follows the one currently being retrieved.

A careful review of ingres documentation revealed that ingres does, at a very low level, support a concept of "next tuple". Further study revealed however, that to achieve the equivalence between the next tuple and the alternate clause, one of two things would have to be done. Either a major rewrite of ingres would have to take place, or all of the ingres security, protection and integrity constraints would have to be by-passed. Neither of these prospects was very appealing. In the first case, ingres source encompasses some 2.5 million lines and a rewrite of even 10% of this is a software task for

a team of programmers with a lot of time available to them. In the second case, the goal of the research was to interface to an existing database system known to support commercial databases. Having its security and integrity constraints by-passed would not be very appealing to any company which valued its data enough to have purchased ingres.

Having eliminated the possibility of interfacing at a very low level, an area of concern was the question of how much of the data in the database should be brought forward for the prolog interpreter to operate on at any given time. There were essentially three possible answers to this question. Whenever a goal name matched a relation name:

1. retrieve the entire relation and store the tuples as instances. This has the appeal that only one call on ingres need be made, even if the same goal appeared multiple times in the body of the parent goal. However, on a large database, this could very quickly use up all of the Atom work area. Remember, Atom memory is not a stack, and removal of all references to a particular ATOM does not cause the ATOM to be removed or Atom work area usage to decrease. In addition, some mechanism would have to be implemented to remove the instances from the Heap when a goal was completed to ensure that space was not taken up by facts which were no longer in use. This mechanism would have to distinguish between instances which were tuples from the database and instances which had been

asserted by the user. ,

2. retrieve and store as instances, only those tuples whose attributes matched the values of any already instantiated arguments in the goal. That is, do a select on the relation before retrieving the tuples. This method should slow the growth of the Atom work area usage as the only new ATOM's introduced are those in attributes which were not instantiated at the time the goal was invoked. However, many instances, and consequently many ATOM's, may be created which are never used. This is because the first instance on which unification is attempted is guaranteed to match. The only reason for retrieving other instances is to cover the possibility that a later goal fails, which would result in a requirement to get a new instance of the current goal. This choice also has the disadvantage that if a given relation name appeared twice in the body of a goal, then there would be no way of determining whether a tuple retrieved for the second goal, which was identical to a tuple retrieved for the first goal, was the same instance or a (possibly desired) duplicate. , or
3. retrieve one (possibly qualified) tuple. This method causes the slowest increase in Atom work area usage as new ATOM's are only introduced as needed. Again, slowing the rate of increase of Atom work area usage is important because there is no mechanism for removing unwanted ATOMS. In addition, since all instance/tuples of the relation are identical in form,

space need only be reserved for one instance/tuple of each relation. We need only reserve space for one instance per relation because, once a tuple/fact is used to solve a goal, we will never want that fact again for that goal. Therefore, we can fill in the arguments of the instance with facts/tuples from the database whenever we require a fact from that relation to solve a goal. The problems associated with this method relate to the way ingres views queries. i.e.: "what is the next tuple that the user requires?". Ingres does not permit nested retrievals. If the user (or user's application using Equel) wants to retrieve a tuple from a relation, retrieve a tuple from another relation, and then get the next tuple from the first relation he must use the first retrieved tuple as a key to put himself in the right place in the relation before he can get the next tuple. We, therefore, have to create a mechanism for marking our place in the database. Despite the ingres "nested retrieval" problems, use of a temporary relation permits this method to be chosen for implementing the Progres system.

Other problems requiring resolution, fall more or less into the areas of general programming. These include:

What view of the system will the user take? That is, is this an enhanced prolog system, an enhanced ingres system, or something completely different?

Given the first viewpoint, what knowledge about the data may the

user be expected to have?

Is the user expected to specially identify queries to the ingres database as opposed to queries on prolog's internal database?

How does the user specify a database he wants to operate with?

How are prolog, ingres and user errors identified/handled?

Internally, do instances which are database tuples have to be handled in some way which is different than the normal unification process?

What process(es) need(s) to be modified?

What new processes need to be written?

5.2 Design

It was decided that the system should be viewed as an enhanced prolog system and that the enhancements should not require any significant changes to the way the prolog user viewed or utilized the system. The only knowledge the user had to have was the name of the database he wished to use and, if he had no other knowledge about the database, that all databases contain two relations called "relation" and "attribute", which could be accessed to gain other necessary information about the database. The user should not have to specify that a goal is a database query.

With reference to the question about the amount of data that should be retrieved, it was decided that the ingres problems were surmountable and that "tuple at a time" retrieval (choice 3) provided the most practical method of implementing the interface. The

mechanism used to mark our place in the database was the use of a temporary relation. After a tuple, from the temporary, is "used", it is deleted so that a later retrieval will recover the "next tuple". Internally, aside from the processes described above, the control processes were to be left undisturbed. At the meta-level, two new predicates, available to the user, (opendb/1 and exitdb/0) were introduced. In addition, a predicate \$cleantmp/0, not available to the user, was created to perform housekeeping chores. These predicates will be described in more detail later.

The design method does not modify the ingres system. It makes use of the Equel pre-compiler to access the database. This implies that any relational database which provided a similar pre-compiler could be interfaced to prolog by simply replacing the Equel statements with the equivalent statements for its pre-compiler.

5.3 Implementation

5.3.1 Overview

Because we could not easily obtain a pointer to a tuple, a different method had to be used to retrieve the tuple and transform it to a form usable in prolog. The first requirement was to identify to prolog the occurrence of an ingres tuple. This was accomplished by creating an instance in which the CLAUSEREC had a special bit set signifying that this was an "ingres instance" (The "ingres fact" from the introduction to chapter III). Next was a requirement to allocate space where the retrieved tuple could be stored until it could be

transformed into a form usable by prolog (strings to ATOMs, numbers to primitives). For this purpose a new structure called an ATTRIBUTE was created and a chain of these built for each relation so that there was one ATTRIBUTE in a chain for each attribute of the relation associated with the chain.

The SKEL in the ingres instance had its first pointer (Fn) changed to point to the chain of ATTRIBUTES associated with the instance. The remaining arguments of the SKEL are vacant until a tuple is actually retrieved.

If we open the database of figure 1-1 and then assert our three rules, memory will look like figures 5-1 through 5-4.

Because of ingres' aversion to nested retrievals, a method had to be conceived to ensure that back-tracking to an ingres instance would always get a new tuple. This was accomplished by causing a temporary relation to be built whenever an ingres instance is encountered as a new instance. For the sake of efficiency, if the invoking goal has some of its arguments instantiated, then only those tuples which have equivalent attribute values are retrieved into the temporary. After a tuple is retrieved from the temporary, it is deleted from the temporary, thus ensuring that later retrievals will always get a new tuple. Remember, this temporary is associated strictly with attempts to solve this goal. The tuple is retrieved, used immediately to instantiate the arguments of the goal, and "forgotten". It will never be used again for this invocation. It is for this reason, that we need only one "ingres instance" associated with a functor. A later

invocation of the same functor will cause a new temporary to be built which will only be accessed to solve goals associated with that invocation.

Let us use our old stand-by, the widget goal, to see how all this works.

Before entering the widget goal, we must first enter the built-in predicate "opendb(parts_db)." to cause the ingres instances of figures 5-1 through 5-4 to be built. Then we must assert the rules to complete the memory layout of the figures.

Now we enter "widget(z), print(Z), nl, fail;.". The same structure as shown in figure 4-1 is again built and the attempt to solve the goal widget(Z) will result in the ingres instance for widget (figure 5-3) being encountered. At this point a command to ingres will cause a temporary to be built and since the widget argument is not instantiated, all the widget tuples are retrieved and placed in it. Now a command to ingres is issued to retrieve the first tuple from the temporary and "w1" appears in the ATTRIBUTE structure. This is used

FIGURE 5-1 - HEAP AND RELSTACK ENTRIES
FOR supplies/2

FIGURE 5-2 - HEAP AND RELSTACK ENTRIES FOR subpart/2

FIGURE 5-3 - HEAP AND RELSTACK ENTRIES FOR `widget/l`

FIGURE 5-4 - HEAP AND RELSTACK ENTRIES FOR `gadget/1`

in a qualifying statement in a command to ingres to delete the tuple from the temporary. "w1" is then "lookup"d (see figure 3-21) to cause an ATOM to be built and a pointer to this ATOM is placed in the Arg position (lvar1 in figure 5-3) in the SKEL of the ingres instance. Since there are more tuples in the temporary, pointers are adjusted so that a later failure will cause progres to use the ingres instance as an alternate clause. Now unification takes place as though the clause "widget(w1)" actually existed as a prolog instance. "w1" will, of course, unify with "2" and the rest of the user's goals will be executed (success #1 in figure 1-5). As before the user's fail goal will cause the interpreter to return to the widget frame for another try at instantiating "2". This will succeed twice more using the ingres instance (successes 2 and 3) before there are no more tuples in the temporary relation.

When all tuples in the temporary have been retrieved, the temporary is destroyed and pointers are adjusted so that the "real" next instance is accessed. At this point the rule "widget(X) :- subpart(X,Y), gadget(Y)." is invoked. When the interpreter tries to solve the subgoals, it will find the pointer to subpart/2 in the body (in the SKEL) of the rule and will, therefore, find the ingres instance for subpart (see figure 5-2). Again a temporary relation will be built and, because there are no instantiated arguments, all the subpart tuples will be retrieved into it. The first retrieval will cause "w4" to appear in the first (subprt) ATTRIBUTE and "g1" to appear in the second (supprt) ATTRIBUTE. The tuple is deleted and

pointers to ATOMs for "w4" and "g1" replace lvar1 and lvar2 (figure 5-2) respectively. These will of course unify with "X" and "Y" and the second goal in the rule body, "gadget(Y)", will be attempted. Note that "Y" is now instantiated to "g1". Following the pointer from the SKEL to gadget/1 (see figure 5-4) the interpreter finds another ingres instance and builds another temporary relation. This time the argument of the invoking goal is instantiated, so only tuples in gadget whose attribute is "g1" are retrieved into the temporary (there is only one). When the tuple is retrieved, and deleted the temporary is empty so it is destroyed. There is now no alternative clause for gadget so the last choice point variable is pointed at the subpart FRAME. However, after "g1" is lookup'd and its pointer placed in lvar1 (figure 5-4), unification will be successful (success #4) and the rest of the user's goals will be executed.

The resulting failure will cause a new tuple to be retrieved from the subpart temporary. This will be "w5,g2" and again a new gadget temporary will be built with just one tuple, "g2", retrieved into it. When "g2" is retrieved, the temporary will be destroyed. "g2" will now be place in the lvar1 slot (figure 5-4) and unification will succeed (success #5). This will again result in the user's goal proceeding to its fail goal.

Another repetition will cause "w6" and "g2" to be retrieved from subpart and "g2" from gadget resulting in another execution of the user's fail goal (success #6). At this point, nothing retrieved from the subpart temporary will cause a successful retrieval into a gadget

temporary (right side of figure 1-5). Therefore, when its temporary is empty, subpart will fail since there is no alternative instance in subpart (figure 5-2). There are no more alternate instances for widget (figure 5-3) either, so the user goal fails completely.

The rest of this chapter uses pseudo code and examples to describe, in more detail, the data structures and the operation of the interface.

5.3.2 Memory and Data Structure Changes and New Data

Structures

Having taken the view that a relation name is equivalent to a predicate name (or Functor) and that the attributes of a tuple are the arguments of an instance, then the only difference to be reconciled between ingres and prolog is the treatment of arguments/attributes. An argument of a given predicate in prolog can be a primitive (number) in one instance, an atom in another, a variable in a third case, and even a compound term in another case. Attributes in an ingres relation, on the other hand, are strongly typed. An attribute of all tuples of a relation must have the same type and cannot exceed a given length. To overcome this problem a new data structure "ATTRIBUTE" was defined. Its declaration is shown in figure 5-5. The comments in the figure explain the usage of the objects in the structure. Figure 5-6 shows a diagrammatic representation of this structure. A chain of these structures is built for each relation in the database (one ATTRIBUTE for each relation attribute). Note that the DATA union is

used only for temporary storage before converting the retrieved object into its prolog form as an ATOM or a constructed Float or Int.

```
typedef struct ATTRIBUTE
/* Every attribute of every relation in an
   open database will have this kind of struct
   associated with it */
Begin
    PTR    Next_Att;
    /*Pointer to the next attribute structure in
       a chain associated with a relation */
    PTR    Att_Name;
    /*ATOM of string which is the name of this
       attribute */
    short  Att_Length; /*max length of
                        domain contents */
    short  Att_Type;   /*permissible :
                        0 - char* , 1 - float, 2 - int */
    union
    Begin
        float AsFloat; /* storage for
                        floating pt. numbers. */
        int   AsInt;   /* storage for integers */
        char  AsString[1];
        /* Note: must be last declaration in struct
           as, in practice, this becomes a variable
           length string */
    End DATA;
End ATTRIBUTE;

#define szofat      sizeof(ATTRIBUTE)
#define Asfloat     DATA.AsFloat
#define Asint       DATA.AsInt
#define Asstring    DATA.AsString
#define TattP(p)    ((ATTRIBUTE*)(p))
```

FIGURE 5-5 - ATTRIBUTE STRUCTURE DECLARATION

A second structure, EXIT_Entry, (shown in figure 5-7) was necessitated so that an exitdb/0 was not executed while a previous goal was accessing the database. A diagrammatic representation of the

FIGURE 5-6 - ATTRIBUTE STRUCTURE DIAGRAM

structure is shown in figure 5-8. Before actually closing the database, exitdb builds a list, using this structure, of all the CLAUSEREC's associated with relations and at the same time checks if they are in use. If any are in use, the exit is aborted and the user advised.

```

typedef struct EXIT_Entry
/* Used by exitdb and cleanup to list
   pointers to clauses needing to be
   erased */
Begin
PTR      E_Next;          /* Next entry */
PTR      Atomptr;         /* Atom entry for
                           relation-name string */
PTR      Functorptr;     /* Functor of relation */
PTR      Clauseptr;       /* Clause to be erased */
short    Arity;           /* Number of attributes
                           in relation */
End EXIT_Entry;

```

FIGURE 5-7 - EXIT_ENTRY STRUCTURE DECLARATION

FIGURE 5-8 EXIT_ENTRY STRUCTURE DIAGRAM

To support the use of these new structures, a new memory area called Relstack was inserted between the Atom work area and the auxiliary stack. A static pointer keeps track of the next available stack area.

Prolog data structures were modified as well. The ATOM has an unsigned char, I_flag, inserted just before the string, which identifies ATOM's which have only occurred as the result of interactions with the database. In the CLAUSEREC, the 4-bit of infofcl has been defined as an indicator of ingres clauses (the "ingres fact" flag). In the FRAME four new objects, Funcptr, Clptr, Temp_Name, and T_Count have been added to accommodate pointers to the FUNCTOR and CLAUSEREC, the name of the temporary relation associated with the FRAME, and a count of the tuples left in the temporary relation respectively.

5.3.3 Process Modifications

Five new code segments and four new procedures have been added to

C-Prolog to create the Progres system. Pseudo code for each section and procedure is shown in figures 5-9 through 5-17. Figure 5-18 shows the modifications made to the initialization file to accommodate the interface. Figures 5-19 - 5-24 show the original pseudo code of chapter IV as modified for the new interface pseudo code sections. These figures are essentially reproductions of the original code shown in chapter IV with new or modified lines identified by "***"

```

1.  Setup_Ingres()
2.  Begin
3.      NULL the qualification string;
4.      Use random number generator to make temp_name
5.      Test database to ensure temp_name is unique
6.      Relation_Name = FUNCTOR_name;
7.      ## range of z is Relation_Name
8.      result = Build_Qualification_String(goal,
                                           *Clauseptr->hdofcl)
9.      switch (result)
10.         Begin
11.             case 0:
12.                 ## retrieve into temp_name (z.all)
13.                 ## where qualification_string
14.                 break; /* some args were instantiated */
15.             case 1:
16.                 ## retrieve into temp_name (z.all)
17.                 break; /* no args instantiated */
18.             case 2:
19.                 return ifail;
20.                 break; /* type mismatch */
21.         End
22.      ## range of r is relation
23.      ## retrieve (tuple_count = r.reltups)
24.      ## where r.relid = temp_name
25.      if (tuple_count)
26.         then Begin
27.             cgf->T_Count = tuple_count;
28.             cgf->Clptr = clauseptr;
29.             cgf->Funcptr = functorptr;
30.             cgf->Temp_Name = temp_name;
31.             return ingres_instance;
32.         End
33.      else Begin
34.         ## destroy temp_name
35.         cgf->T_Count = cgf->Temp_Name = NULL;
36.         return ifail;
37.     End
38. End
39. End

```

FIGURE 5-9 - RETRIEVING QUALIFYING TUPLES INTO A
TEMPORARY RELATION

```

1.  get_instance()
2.  Begin
3.      target_ptr = &targ_string;
4.      gipl = *(clauseptr->hdofcl) /* point at the
                                   /* attribute chain*/
5.      while (gipl)
6.      Begin
7.          target_ptr = "%(gipl->Att_Type)=
                        t.((ATOM*)(gipl->Att_Name)->stofae)";
                        /* The above actually requires a number of
                        /* operations but the sense should be clear */
8.          param_array[n] = &(gipl->DATA);
9.          n++;
10.         gipl = gipl->Next_Att;
11.     End /* of loop */
12.     NULL terminate targ_string;
13.     ## range of t is cgf->Temp_Name;
14.     ## retrieve (targ_string,param_array)
15.     gipl = *(clauseptr->hdofcl);
16.     pipl = clauseptr->hdofcl;
    /* Build a qualification string to delete the tuple
    /* and convert the data just retrieved for prolog
    /* use */
17.~  while (gipl)
18.      Begin
19.          qual_ptr =
                "t.((ATOM*)(gipl->Att_Name)->stofae) =
                (gipl->DATA) and ";
20.          Bump(qual_ptr); /* to end of string for next
                           /* part of string */
21.          case (gipl->Att_Type)
22.          Begin
23.              string:***pipl = BuildAtom(gipl->DATA);
24.              break;
25.              Integer:***pipl = conv_prolog_int(gipl->DATA);
26.              break;
27.              Float:***pipl = conv_prologflt(gipl->DATA);
28.              break;
29.          End

```

FIGURE 5-10 - RETRIEVING TUPLES FROM TEMPORARY
(continued next page)

```

30.      gipl = gipl->Next_Att;
31.  End
32.  ## delete t where qual_string
33.      cgf->T_Count--;
34.      if (cgf->T_Count) faillabel = cgf->clptr;
35.      else Begin
36.          faillabel = &(clauseptr->altofcl)
37.  ##          destroy cgf->Temp_Name
38.          cgf->Temp_Name = NULL;
39.      End
40.      return alternate_instance;
41. End

```

FIGURE 5-10 - RETRIEVING TUPLES FROM TEMPORARY

```

1.  case opendb_;
2.  Begin
3.      check that argument is a valid database name
4.      if (database already open)
5.          if (argument == current database name)
6.              return continuation;
7.      else return gfail;
8.  ## ingres argument
9.      make_ingres();
10.     return continuation;
11. End

```

FIGURE 5-11 - PSEUDO CODE FOR OPENING A DATABASE

```

1. case exitdb_:
2.   Begin
3.     PTR startptr, exitptr, atomptr;
4.     int NumberAtts;;
5.     char RELID[13];
6.     startptr = exitptr = 'relstack avail space'
7.     ## range of r is relation
8.     ## retrieve (RELID=r.relid, NumberAtts=r.relatts)
9.     ##      where r.relid != "a temporary"
10.    Begin /* retrieve translates to "while
        /* a tuple is retrieved" */
11.      atomptr = lookup(RELID);
12.      functorptr = atomptr->fcofae;
13.      while (functorptr &&
        (functorptr->arityoffe != NumberAtts))
14.        functorptr = functorptr->nxttoffe;
15.      if (functorptr)
16.        clauseptr=functorptr->FClause.First;
17.      else stop(); /* something has messed up */
18.      while (clauseptr && (!(IsIngres(clauseptr))))
19.        clauseptr = clauseptr->altofcl;
20.      if (!(clauseptr)) stop();
21.      if (In_Use(clauseptr)) return gfail;
22.      at exitptr build an EXIT_Entry and link it
23.      Bump(exitptr);
24.    End /* of retrieve loop */
25.    cleanup(startptr);
26.    return continuation;
27.  End

```

FIGURE 5-12 - PSEUDO CODE FOR EXITING A DATABASE

```
1. case cleantmp_:
2.   Begin
3.     if (not an open database) return continuation;
4.     ## range of r is relation
5.     cflag = FALSE;
6.     while (!(cflag))
7.       Begin
8.         ## retrieve (RELID=r.relid)
9.         ##       where r.relid == "a temporary"
10.        Begin
11.          set(cflag);
12.        End
13.        if (cflag)
14.          Begin
15.            ## destroy RELID
16.            reset(cflag);
17.          End
18.        else set(cflag);
19.        return continuation;
20.      End
21. End
```

FIGURE 5-13 REMOVING TEMPORARY RELATIONS FROM
THE DATABASE

```

1. make_ingres()
2. Begin
3. PTR lptr;
4.   if (there is a progres system relation)
5.   ##   destroy prosys
6.   ## range of r is relation
7.   ## range of a is attribute
8.   ## retrieve into prosys (r.relid,r.relatts)
9.   ##   where r.relid != "a temporary"
10.  ## range of p is prosys
11.   reset(readyflag);
12.   while (!(readyflag))
13.   Begin
14.   ##   retrieve (RELID=p.relid,NumberAtts=p.relatts)
15.   ##   where p.relid != "an admin relation"
16.   Begin
17.   functorptr =
18.   FUNCTOR_RELATION(RELID,NumberAtts);
19.   clauseptr = record(CLAUSE,functorptr,0,1);
20.   clauseptr->infofcl = INGRES;
21.   Set(readyflag);
22.   break;
23.   End
24.   *(clauseptr->hdofcl)=attptr=available space
                                   in relstack
25.   if (readyflag)
26.   Begin
27.   reset(readyflag)
28.   ##   delete "the tuple just retrieved"
29.   ##   retrieve (name = a.attname,
30.   ##             type=a.attfmt,
31.   ##             length = a.attfrml)
32.   ##   where a.attrelid == RELID
33.   Begin
34.   Build and ATTRIBUTE and link it
35.   End /* of retrieve loop */
36.   End
37.   else set(readyflag);
38.   End /* while loop */
39.   ## destroy prosys
40.   return
41. End

```

FIGURE 5-14 - CREATING THE DATABASE ENVIRONMENT

```
1. cleanup(E_Ptr)
2. Begin
3.   while (E_ptr)
4.     Begin
5.       gipl = (E_Ptr->Clauseptr)->hdofcl;
6.       *gipl = E_Ptr->Functorptr;
7.       erase(E_Ptr->Next);
8.       E_Ptr = E_Ptr->Next;
9.     End
10. ## exit
11.   INGRES_NAME = NULL;
12.   reset(available space pointer in relstack
13.         to start of relstack);
14.   return;
15. End
```

FIGURE 5-15 - REMOVING THE DATABASE ENVIRONMENT

```

1. Build_Qualification_String(goal,Atts)
2. PTR goal,Atts
3. Begin
4.   qual_ptr = qual_string;
5.   while (Atts)
6.     Begin
7.       gip1 = *++goal;
8.       dereference gip1 using gip2 and gip3 as in
                                     range
9.       if (gip2 is a composite term or
           gip2 is a variable)
10.      then Begin
11.        Atts = Atts->Next_Att;
12.        continue;
13.      End
14.      if (type(gip2) != Att->Att_Type) return 2;
15.      qual_ptr =
        "z.((ATOM*)(Atts->Att_Name)->stofae) =
        (Atts->Data) and";
16.      Atts = Atts->Next_Att;
17.      Bump(qual_ptr);
18.    End
19.    if (there were any qualifying tuples)
20.    then Begin
21.      NULL terminate qual_string;
22.      return 0;
23.    End
24.    return 1;
25. End

```

FIGURE 5-16 - QUALIFYING STRING FOR RETRIEVE INTO
TEMPORARY

```
1. FUNCTOR_RELATION/Relname,NumberAtts
2. Begin
3.   Build a string of the form
      Relname(X1,X2,...NumberAtts).
4.   return term(string);
      /* This is a call on prolog
      /* procedures to parse "string"
5. End
```

FIGURE 5-17 - BUILD FUNCTOR's AND ATOM's FOR
DATABASE RELATIONS

```

1. test_alternatives()
2. Begin
3.   if (IsIngres(clauseptr) && (!(cgf->Temp_Name)))
4.     return ingres_setup;
5.     if (IsIngres(clauseptr)) return ingres_instance;
6.     faillabel = &clauseptr->altofcl;
7.     return alternate_instance;
8. End

```

FIGURE 5-21 - TEST_ALTERNATIVES (FLOW CONTROL)
PROCEDURE

```

1. alt()
2. Begin
3. *****
4.   if (!(*faillabel)) lcp = .cgf->altofcl;
5.   clear space after current FRAME for local and
6.   temporary variables
7.   do the same for global variables in global
8.   cgf->altofcl = faillable;
   /* at this point the predicate name has been matched
   if there are no arguments, check for a body */
9.   if (IsAtomic(goal)) return body_check;
10.  return resolve_args;
11. End

```

FIGURE 5-22 - MODIFIED ALTERNATE INSTANCE HANDLING

```

$assert(C):-$assertr(C),!.
$assert(C):-
    seeing(user),
    !.
$assert(C):-
    display('! clause: '),
    display(C).
$directive(:-(X),_,X,command):-!.
$directive(?-(X),_,X,question):-!.
$directive(X,0,X,question).
$dogoal(command,C):-
    $user_exec(C),
    !.
$dogoal(command,_) :-
    !,
    ttynl, display('?'),
    ttynl.
$dogoal(question,Q):-
    $save_read_vars,
    $user_exec(Q),
    $reply,
    ttynl, display(yes),
    $reset_read_vars,
    ttynl,
    !.
$dogoal(question,_) :-
    ttynl, display(no),
    $reset_read_vars,
    ttynl,
    !.
$opendb(X):-$opendb(X).
$opendb(X):171.
$exitdb:-$exitdb.
$exitdb:-172.
$cleantmp:-173.

```

FIGURE 5-18 - MODIFIED INITIALIZATION FILE

```
1. While (TRUE)
2.   Begin
3.     switch (switchvalue)
4.       Begin
5.         interpret_goal: switchvalue = ical1();
6.         break;
7.         alternate_instance: switchvalue = alt();
8.         break;
9.         body_check: switchvalue = bcheck();
10.        break;
11.        resolve_args: switchvalue = rargs();
12.        break;
13.        eval_pred: switchvalue = epred();
14.        break;
15.        body_exists: switchvalue = ebody();
16.        break;
17.        continuation: switchvalue = carry_on();
18.        break;
19.        no_body: switchvalue = neck_foot();
20.        break;
21.        ifail: switchvalue = instance_failed();
22.        break;
23.
24.        gfail: switchvalue = goal_failed();
25.        break;
26.**   ingres_setup: switchvalue = setup_ingres();
27.        break;
28.**   ingres_instance: switchvalue = get_instance();
29.        break;
30.**   test_alt: switchvalue = test_alternatives();
31.        break;
32.     End /* of switch */
33.
33. End /* of loop */
```

FIGURE 5-19 - MAIN CONTROL LOOP MODIFIED FOR INTERFACE

```

1.  icall()
2.  Begin
3.      if (pg points to a SKEL) /* actually determined */
                                   /* by relative addressing */
4.      then goal = pg; /* this is the address of an ATOM
                                   /* or the address of a gtoffe of a FUNCTOR */
5.      else Begin
6.          pglbf = ((MOL*)(pg))->Env;
7.          goal = ((MOL*)(pg))->Sk;
8.      End
9.      functorptr = *goal;
10.     clauseptr = functorptr->FClauses.First;
/* Because FUNCTOR's and ATOM's are identically
structured in the first 7 words, they can be treated
identically for the objects in those 7 words. In
addition, it doesn't matter whether goal contains the
address of a gtoffe or the address of an ATOM because
the contents at that address will be the address of the
appropriate structure. If the structure is an ATOM
however, clauseptr may now contain an integer
representing the case label of an atomic built-in
predicate such as !(cut) or abort. */
11.    if (0 < clauseptr < 255)
12.    then Begin
13.        case_label = clauseptr; return eval_pred;
14.    End
/* CLAUSEREC's which have been retract'd are not
immediately unlinked. Instead, a bit in the CLAUSEREC
is set and a garbage collection algorithm later cleans
up the Heap */
16.    while (clauseptr && erased(clauseptr))
17.        clauseptr = clauseptr->altofcl;
18.    if (!clauseptr) return gfail;
/* build a frame */
19.    cgf->lcpofcf = lcp;
20.    cgf->gofcf = pg;
21.    cgf->gfofcf = pgf;
22.    cgf->gsocf = glbf;
23.    lcp = cgf;
24.** if (IsIngres(clauseptr)) return ingres_setup;
25.    return alternate_instance;
26. End

```

FIGURE 5-20 - MODIFIED PREDICATE RESOLUTION PROCEDURE

```

1. test_alternatives()
2. Begin
3.   if (IsIngres(clauseptr) && (!(cgg->Temp_Name)))
4.     return ingres_setup;
5.   if (IsIngres(clauseptr)) return ingres_instance;
6.   faillabel = &clauseptr->altofcl;
7.   return alternate_instance;
8. End

```

FIGURE 5-21 - TEST_ALTERNATIVES (FLOW CONTROL)
PROCEDURE

```

1. alt()
2. Begin
3.   *****
4.   if (!(*faillabel)) lcp = .cgg->altofcl;
5.   clear space after current FRAME for local and
6.   temporary variables
7.   do the same for global variables in global
8.   cgg->altofcl = faillabel;
9.   /* at this point the predicate name has been matched
10.  if there are no arguments, check for a body */
11.  if (IsAtomic(goal)) return body_check;
12.  return resolve_args;
13. End

```

FIGURE 5-22 - MODIFIED ALTERNATE INSTANCE HANDLING

```
1. instance_failed()
2. Begin
3.   clear local and global variables; /* some may
      /* have been instantiated
      /* before the instance
      /* failed*/
4.   clauseptr = *faillabel;
5.   while ( clauseptr && erased(clauseptr))
6.     clauseptr = clauseptr->altofcl;
7.**  if (clauseptr) return test_alt;
8.   lcp = cgf->lcpofcf; /* this goal failed
      /* completely so find the last
      /* choice point and cause a
      /* "fail back" to the last
      /* parent having a choice
      /* point */
9.   return gfail;
10. End
```

FIGURE 5-23 - MODIFIED INSTANCE FAILURE HANDLING

```

1.  *goal_failed()
2.  Begin
3.      while (TRUE) /* find a choice point FRAME having
                       /* a CLAUSEREC that is not erased */
4.      Begin
5.          for (clauseptr = *(lcp->altofcl);
6.              clauseptr && erased(clauseptr);
7.              clauseptr = clauseptr->altofcl);
8.          if (clauseptr) break;
9.          lcp = lcp->lcpofcf;
10.     End
11.     clear local and global variables;
12.     /* reset pointers */
13.     cgf = lcp;
14.     pgf = cgf->gfofcf;
15.     pg = cgf->gofcf;
16.     cont = cgf->cofcf;
17.     if (pg points to a SKEL)
18.     then Begin
19.         goal = pg;
20.         pglbf = pgf->gsocf;
21.     End
22.     else Begin
23.         goal = ((MOL*)(pg))->Sk;
24.         pglbf = ((MOL*)(pg))->Env;
25.     End
26. ** return test_alt;
27. End

```

FIGURE 5-24 - MODIFIED GOAL FAILURE HANDLING

5.3.4 Meta-level Modifications

Figure 5-18 shows the new built-in predicates and the modified meta-level interpreter. The built-in predicate, `exitdb` has been used in the `$cycle/1` goal to ensure that Prolog is not exited without first exiting the database if one is open. Figure 5-12 shows the pseudo code for the case where `exitdb/0` is executed. If there is no database open, it simply returns continuation. This pseudo code will be examined in more detail later.

In the third `interpret/2` goal a new predicate `$cleantmp/0` has been added. This is a new built-in predicate which is not available to the user. When a user goal succeeds, this goal will be executed. Figure 5-13 shows the pseudo code for this goal. If a database is open, then it will automatically remove any temporary relations which may have been left as a result of the goals success. This code will also be reviewed in detail later.

5.4 Operation of the Interface

The following examples will demonstrate the operation of the interface using the `ingres` database of figure 1-1. Note that rules cannot be represented in an `ingres` data-base. They must, therefore, be stored in a separate file which the user causes `prolog` to consult.

Example 1. Before the facts in the database can be used `Prolog` must be "told" about the database. This is accomplished by entering the following command.

```
opendb(parts_db).
```

When `$dogoal/2` calls `$user_exec/1`, the goal of `opendb ($opendb(X))` will unify resulting in case number 171 (figure 5-11) (in `epred()`) being executed. The check at line 3 of figure 5-11 just ensures that the name is a string. `Ingres` does not permit two databases to be open at once, so, a check is made to ensure that there is not already a database open under another name (lines 4 and 5). If there is then the predicate fails. Otherwise, at line 8, `ingres` is invoked and the database is opened. At line 9 the procedure "`make_ingres()`" is called

to establish the proper environment before a "return continuation" is executed.

In `make_ingres` (figure 5-14) the database is first checked to ensure that some previous abnormal termination has not left a Progres system relation (`prosys`) in the database. If there is a `prosys` relation, it is removed (line 5).

In any ingres database, there are always two ingres system

relid	relatt
relation	13
attribute	8
supplies	2
subpart	2
gadget	1
widget	1

attrelid	attname	attfmt	attfrml
relation	relid	c	12
relation	reltups	i	4
relation	relatts	i	2
attribute	attrelid	c	12
attribute	attname	c	12
attribute	attfmt	c	1
attribute	attfrml	i	1
supplies	sname	c	4
supplies	pname	c	2
subpart	subprt	c	2
subpart	superprt	c	2
gadget	gpart	c	2
widget	wpart	c	2

FIGURE 5-25 - SELECTED PARTS OF `partadb` SYSTEM RELATIONS

relations called "relation" and "attribute". Figure 5-25 shows attributes of interest to Progres from these relations. Progres needs the name of every relation and the number of attributes in that relation. Also a description of the type, length, and name of each attribute of every relation is needed. At lines 6 and 7 (figure 5-14), the ingres variables "r" and "a" are declared to range over "relation" and "attribute" respectively. At line 8, the "prosys" relation is created with the attributes "relid" and "relatts" projected from "relation". If a temporary relation was left on the database from a previous abnormal termination it does not appear in "prosys" (line 9). At line 10, the ingres variable "p" is declared to range over "prosys".

As mentioned previously, ingres does not permit nested retrievals, so a loop is entered at line 12. In the loop, a tuple is retrieved from "prosys" and the retrieved name and number of attributes is passed to FUNCTOR_RELATION() (figure 5-17). There, a string is built having the form "relation_name(X1,X2,...Xnumber of attributes)". This is then parsed, ATOM's and FUNCTOR's are built, and a pointer to the MOL returned. At line 19 of figure 5-14, the prolog procedure "record" is called which creates a CLAUSEREC and instance for the string. The 4-bit of infofcl in this CLAUSEREC is set (line 20) to specify that this is an ingres instance. A flag is set (line 21) indicating that a relation name was returned and the retrieve loop is exited.

At line 24 memory pointers are initialized for a new attribute

chain and at line 28, the current tuple is deleted from "prosys". Note that, at line 24, the SKEL->Fn is set to point to the space where the attribute chain is about to be built. Now a new retrieval loop on "attribute" (lines 29-35) is entered to retrieve in successive ATTRIBUTE structures, the necessary information about each attribute for the current relation. When this is complete the control returns to line 12 and a new tuple is retrieved from "prosys". When "prosys" is "empty" the loop is exited and "prosys" is destroyed. This code could have been written more simply if the retrieval into "prosys" had read "retrieve into prosys (r.relid, r.relatts, a.attname, a.attfmt, a.attfml) where a.attrelid == r.relid". However, on a retrieve into a temporary, ingres sorts the tuples on the keys of the base relations. This results in the attributes not appearing in "prosys" in the same order in which they occur in the relation. This could be needlessly confusing to the user. Since this code is only executed once, when the database is opened, it makes sense to use this slower but less confusing method.

This will complete the opendb/1 predicate and \$dogoal/2 will complete its execution. Note that at the meta-level the next goal in the third \$interpret/2 is \$cleantmp.

When \$cleantmp/0 is executed, case clntmp_ (figure 5-13) interpreted() is called. There is now an open database so the return in line 3 is not executed. "r" is again declared to range over "relation" and a loop similar to the one in make_ingres() is entered at line 6. Temporary relations are identified by the fact that the first

character of the users login id occurs as the first character of the relation name and the next five characters of the login id appear as characters 8 through 12 with a random number in the intervening 6 character positions. In the `clntmp_code`, therefore, a retrieval is made on "relation" where `relid == "c?????ccccc"` (lines 8 and 9). If the retrieval is successful a flag is set and the retrieved relation name is destroyed. Again, this awkward construct is necessitated by `ingres` refusal to execute nested commands. Note there is no need to delete tuples from "relation" (in fact it is impossible) because the destruction of the temporary relation causes its associated tuple in "relation" to be automatically deleted.

When any temporary relations have all been destroyed, `clntmp` (`epred()`) "returns continuation". Thus `$interpret/2` continues resulting in its failure and the `$repeat/0` in `$cycle/1` gets reexecuted causing a new prompt to be displayed and the `$read/1` to await terminal input.

Example 2. In this example, suppose `opendb/1` was executed, a prolog program run, and now the following predicate is entered, "exitdb.". This predicate causes case `exitdb_` (figure 5-12) in `epred()` to be executed. In figure 5-6, pointers are initialized to the next available space in `relstack` (line 6). A retrieval loop is entered where the names and number of attributes of all non-temporary relations are retrieved. Using the `ATOM` of the `relation_name` (lines 11 and 12) as a starting point, the `FUNCTOR` chain is searched for a

FUNCTOR with an arity equal to the number of attributes (lines 13 and 14). When the FUNCTOR is found, the (ingres) CLAUSEREC is searched for (lines 18 and 19). Inability to find the FUNCTOR or CLAUSEREC implies an internal error and the prolog procedure "stop()" is called (lines 16 and 20) to cause a memory dump.

Finding the CLAUSEREC to be in use, implies a user goal of the form "relation_name(X,a,...),exitdb.". If exitdb is allowed to succeed under these conditions and the user then enters ";<cr>", "relation_name" may attempt to access the closed database resulting in an ingres error. Therefore, exitdb/0 fails (line 21) if an ingres CLAUSEREC is found to be in use when exitdb/0 is executed.

If the CLAUSEREC is not in use, an EXIT_Entry structure is linked into a chain in relstack for later processing (lines 22 and 23). The loop continues until all relation_names in the database have been retrieved. Then, before executing "return continuation", the procedure cleanup() (figure 5-15) is called (line 24) with a parameter which is a pointer to the start of the chain of EXIT_Entry structures.

In cleanup(), the Fn object in the SKEL is reset to point to the associated FUNCTOR so that the prolog "erase" procedure will correctly erase the ingres instance. After all the ingres instances have been erased, ingres is exit'd. To complete the process the variable "INGRES_NAME" which contains a pointer to the ATOM of the database name is NULL'd to indicate there is no longer a database open and relstack pointers are reset so that all of relstack becomes available.

Example 3. (The big one that ties it all together.) As a final example, suppose we have the ingres database of figure 1-1 and the prolog "rules_file" of figure 5-26.

```
supplies(acme,X) :- widget(X).
supplies(X,Y) :- subpart(Y,Z), supplies(X,Z).
widget(X) :- subpart(X,Y), gadget(Y).
```

FIGURE 5-26 - RULES_FILE

Now suppose the following goals were entered successively at the terminal.

```
consult(rules_file).
opendb(parts_db).
supplies(X,Y),widget(Y),print(X),nl,print(Y),nl,
fail.
supplies(bar,X),print(X),nl,fail.
exitdb.
```

These may be thought of as meaning:

"Get the rules into memory."

"Open the database."

"Who supplies what widgets?"

"What does 'bar' supply?"

"Exit the database."

After the first two predicates have executed, memory will appear as shown in figures 5-1 through 5-4.

When the third goal is executed and parsed a complicated

structure is built in global memory to control the order of execution of the the goals. This looks somewhat like figure 4-16 but its structure is not germane to describing how the new procedures work, and will not be further discussed except to note that some action has taken place to cause a pointer to change value.

After \$user_exec/1 is executed and the pointers have been adjusted, pg is left pointing to a "SKEL like" structure in global memory which contains as its first object a pointer to the gtoffe of supplies/2 and whose second and third words contain pointers to vacant areas in global memory reserved for the variables "X" and "Y". At this point icall() (figure 5-20) is called from the main loop.

Since pg is not pointing to a skel, lines 6 and 7 get executed followed by 9 and 10. Thus pglbf points to the area reserved for "X" and goal points to the gtoffe of supplies/2. Clauseptr points to the ingres CLAUSEREC (see figure 5-1). After the FRAME is built, in lines 19-23 the return ingres_setup of line 25 is executed which causes the main loop to call setup_ingres() (figure 5-9).

In setup_ingres a random number generator inserts a 6 digit number into the temporary name array and the database is checked to make sure that it is unique (lines 4 and 5). Then the variable "z" is declared to range over the supplies relation and Build_Qualifying_String (figure 5-16) is called. In this procedure the attribute chain pointed to by the Fn entry in the ingres SKEL is used to build up a string of the form "z.att1=arg1 and z.att2 = arg2 and ..." where a given att/arg combination is included only if the arg

collection mechanism could go in and unchain all ATOMs which were strictly ingres ATOMs. Then the save and restore procedures could be used to compact and relocate the ATOMs.

The most glaring deficiency in the interface is speed of operation. The prolog partsfile of figure 1-4 was used as the database for testing the speed of C-Prolog while the partsdb of figure 1-1 and the rules file of figure 5-26 were used to test the speed of execution of Progres. The following query was entered:

```
"cputime(X),print(X),nl,
  widget(Y),
  cputime(Z),
  print(Y),nl,
  print(Z),nl;
fail."
```

"cputime" is a built in predicate which returns the number of cpu seconds since the interpreter was executed. This very simple query took .1 seconds to execute under C-Prolog. Progres took .5 seconds to execute. This may not be as bad as it appears. While Progres does take longer, it does do a "select" on the relation of interest if a goal has some of its arguments instantiated. There is no such selection mechanism available for reducing the search path in prolog. This implies, that Progres should be more efficient if a large database is involved and some of the arguments are instantiated when the system tries to solve a goal.

There are a number of ways in which the speed might be improved.

string of the form "%c = attribute_name, %i2 = attribute_name,...." followed by the name of the array. The % is an identifier that the following letter and, optionally a number, is the type and length of the storage location whose address is stored in the corresponding element of the array. Because the prolog interpreter never "knows" apriori what the type of the variables will be, this form of retrieval is used.

In `get_instance()` the target string is built up using the attribute chain to determine the attribute names, and the addresses of the DATA areas are placed in the array (lines 5 - 11). Since all tuples in the temporary are already qualifying, there is no requirement to build a qualification string. In this case, the tuple (foo,pl) is retrieved (line 14). Now (lines 18 - 31) the information just retrieved is used to build a qualification string in preparation for deleting the tuple from the temporary. At the same time, the data is converted to prolog form. That is, ATOM's are built for "foo" and "pl" and pointers to those ATOM's are placed in the appropriate "Args" of the ingres SKEL. At line 32, the tuple is deleted. T_Count is decremented and because there are still three tuples left faillabel is adjusted to point to `cgf->Clptr`, which contains the address of the the ingres CLAUSEREC (line 34). `Alternate_instance` is returned (line 40). At this point the system is setup as though an instance "supplies(foo,pl)" existed in the SKEL in the supplies/2 ingres instance and unification of the goal `supplies(X,Y)` will succeed. This causes an attempt to be made to match the goal `widget(Y)` where `Y = pl`.

Widget/1 has as its first instance (see figure 5-3) another ingres CLAUSEREC, so `setup_ingres` (figure 5-9) is again called to build a temporary relation with tuples having "pl" as its sole attribute. There are none, so `ifail` will be returned (line 37) after the temporary is destroyed. `Instance_Failed` (figure 5-23) operates as it did in chapter IV, except that when a new instance is found it returns `test_alt` instead of `alternate_instance`. This causes the main loop to call `test_alternatives` (figure 5-21) to determine whether the instance is a new ingres clause which needs to be set up (line 3), an ingres clause which has already been setup (line 5), or an ordinary prolog clause. In this case the new instance is the prolog rule "widget(X) :- subpart(X,Y),gadget(Y)" so `alternate_instance` (line 7) is returned. Since the argument of widget in the rule is a variable, it will unify successfully with "pl" in the parent and an attempt to satisfy "subpart(pl,Y)" will be made. Again, there is an ingres instance (see figure 5-2) for subpart/2 so `setup_ingres()` will be called to set up a temporary relation of subpart tuples having "pl" as its first attribute. None match, so the temporary is destroyed and `ifail` returned. There are no more subpart instances, so `ifail` will return `gfail`. There are also, no more widget instances so we failback to the supplies FRAME via `test_alternatives`. In this case, `test_alternatives` finds that we still have tuples in the supplies temporary relation (figure 5-21, line 5), so `get_instance()` is called to retrieve (bar,pl):

The same routine is again gone through and again it fails when

subpart cannot match on "pl". These attempts continue all through the supplies temporary until eventually the supplies ingres instance fails. This causes the temporary to be destroyed and a new prolog instance is found (see figure 5-1) ("supplies(acme,X) :- widget(X)"). Since both arguments of the user's goal are uninstantiated, the head of the rule will successfully unify and an attempt will be made to solve widget(X) where X is uninstantiated. This time when the ingres instance for widget is encountered (figure 5-3) and setup_ingres() is called, there is no qualification string, so the temporary will contain all three widget tuples. Thus, setup_ingres() will return ingres_instance (line 31, figure 5-9) and get_instance() will be called to retrieve the "w1" instance, which will successfully unify with widget(X). Now the user's goal "supplies(X,Y)" has succeeded with X=acme and Y=w1. Now the user's goal "widget(Y)" with Y=w1 must be attempted. It will immediately encounter the ingres instance (figure 5-3) and setup_ingres() will be called to build a temporary from the widget_relation where the attribute equals "w1". This will, of course, succeed. Note: At this point there are two temporary relations in the database. Both are associated with widget. The first contains "w2" and "w3" which are alternate facts for "widget(X)" in the "supplies(acme,X)" rule. The other contains only "w1" which will be retrieved for unification with the current user's goal "widget(w1)". Get_instance() will be called to retrieve the instance. This will cause the temporary to be emptied and destroyed but unification will succeed, causing "acme" and "w1" to be printed and

the user's goal "fail" to be executed so that "acme" and "w2" etc can be printed.

If the process is followed to its ultimate failure the results show that "acme" supplies "w1", "w2", "w3", "w4", "w5", and "w6" and that "foo" supplies "w4". This is a rather surprising result when one notes that the widget relation shows only "w1", "w2", and "w3" as widgets and that the supplies relation doesn't show any supplier for any of the widgets.

The supplies(bar,X) user goal can be demonstrated in a similar manner to the first goal. Its results show that in addition to "p1", "bar" supplies "p2" and "p3".

5.5 Conclusion

The Progres system incorporates a transparent interface between C-Prolog and ingres. Data in the database appears to the user as though tuples were ground unit clauses of the form `relation_name(att1,att2,...)`. The interface uses a "select" on the relation of interest to retrieve, into a temporary relation, tuples which have attribute values equal to the values of any already instantiated arguments in the invoking goal. The resulting temporary is accessed as needed to get new facts satisfying the goal. Many temporary relations may be extant at any time depending on the number of user's goals which are being unified with "ingres facts". Two new built-in predicates have been added for user use to enable and disable access to the database.

This interface provides retrieval capabilities only. No attempt has been made to provide any of the other ingres commands. This limitation and others are analyzed in chapter VI.

Chapter Six

ANALYSIS AND CONCLUSIONS

6.1 Analysis

Some rudimentary analysis of the interface has been done using the parts database example. The analysis concerned itself primarily with ensuring correct operation but response times were also determined. The system appears to be both safe and complete in the sense that it produces only demonstrably correct answers and that it produces all the possible solutions to a given goal. There is a curious anomaly in the responses in that, for an undetermined reason, it appears to retrieve tuples in the reverse order from the expected. Figure 6-1 shows the results of entering the Prolog query "widget(X)" with no rules in the system, followed by the result of entering the Prolog query "retrieve (w.wpart)". For most Prolog programs, the order of response on ground unit clauses is unimportant as long as the order of evaluation of facts and rules is not disrupted and that ordering is maintained in Prolog.

For the sake of simplicity, no attempt was made to give the user a choice as to the location of Prolog clauses relative to other Prolog clauses. When the database is opened all Prolog clauses are asserted as the first in the chain. The user can overcome this ordering by judiciously asserting clauses he wants in front of the Prolog clauses using asserta/1 after opening the database. The

```
| ?- widget(X),print(X),nl,fail.
```

```
w3
```

```
w2
```

```
w1
```

```
no
```

```
* range of w is widget
```

```
* retrieve (w.wpart)
```

```
* \g
```

```
Executing . . .
```

wpart
w1
w2
w3

```
(3 tuples)
```

FIGURE 6-1 - REVERSED RETRIEVAL ORDERING

incorporation of the option to have ingres clauses asserted at either end of appropriate chains is not considered to be a difficult task.

Abnormal termination of Progres does not, in general, cause database problems. There may be temporary relations left in the database but the next invocation of Progres, by the same user, will cause those to be removed as soon as the database is opened, since Scleantmp will be executed as a result of the successful open. However, this feature can cause another problem if the user stops Progres (using control z, for instance) while he is executing an "ingres" goal. If he then starts a new Progres and opens the same database, all the temporaries in use by the first invocation will be

wiped out. If he later restarts the first invocation, it will fail as soon as an attempt is made to access a temporary relation that is supposed to be in the database.

The prolog "erase" procedure uses the Fn object in the SKEL's of instances when they are retract'd to determine whether the instance is protected from erasure. Since the FN object is modified in ingres instances to point to the attribute chain, an attempt to "retract" an ingres clause could cause erratic behavior. No attempt has been made to protect against this. There are two ways that it might be done. One is to make the ATTRIBUTE structure appear similar enough to a FUNCTOR that appropriate bits could be set to prevent erasure. The other, and perhaps simpler method, would be to have erase check the CLAUSEREC to see if it is an ingres clause. If it is, then simply do not erase it. Using this method the Progres "cleanup()" procedure would have to reset the ingres bit before calling erase to remove the CLAUSEREC's.

C-Prolog has a second level unification algorithm, called gunify (for global unify), to permit unification on compound terms when they appear as arguments. For example if unification were attempted on the clause "supplies(acme, subpart(X,pl)):", then the second level algorithm would be called to handle unification on subpart(X,pl). Progres does not support this case where subpart is an ingres instance. Though no direct investigation has been done to determine the feasibility of doing this, the second level algorithm seems similar enough to the top level version that implementation of

algorithms like `setup_ingres` and `get_instance` (figures 5-9 and 5-10) should not be difficult.

Saving a Progres program while a database is open is directly inhibited. This was originally done because there could, at the time, be no guarantee that the database would be in the same state when the program was restarted as it was when saved. This could now be relaxed to the extent that a save could be done if there were no relations in use. However, both the "save" and "restore" procedures would have to be rewritten. "Save" would have to save the name of the database and the attribute structures and "restore" would have to handle relocation of attribute structures in a manner similar to what it does now for other structures.

A problem that is common to both C-Prolog and Progres is the fact that ATOM's are stored in a work area for which there is no garbage collection mechanism. This problem is exacerbated in Progres. When Progres opens a very large database, there is an immediate increase in the number of ATOMs as all the names of relations are read in. As many calls are made on the database, more and more ATOM's are created to handle the strings from the retrieved tuples. One way to overcome this problem is to mark ATOMs which have only ever been referred to as ingres attributes. This has already been done in Progres. Because of the fact that ATOMs are not a fixed size they cannot simply be removed or written over. If, when `$cleantmp/0` was invoked, the code checked the usage of the Atom work area, it could determine if usage had exceeded some predetermined threshold. If it had, a garbage

collection mechanism could go in and unchain all ATOMs which were strictly ingres ATOMs. Then the save and restore procedures could be used to compact and relocate the ATOMs.

The most glaring deficiency in the interface is speed of operation. The prolog partsfile of figure 1-4 was used as the database for testing the speed of C-Prolog while the partsdb of figure 1-1 and the rules file of figure 5-26 were used to test the speed of execution of Progres. The following query was entered:

```
"cputime(X),print(X),nl,
  widget(Y),
  cputime(Z),
  print(Y),nl,
  print(Z),nl;
fail."
```

"cputime" is a built in predicate which returns the number of cpu seconds since the interpreter was executed. This very simple query took .1 seconds to execute under C-Prolog. Progres took .5 seconds to execute. This may not be as bad as it appears. While Progres does take longer, it does do a "select" on the relation of interest if a goal has some of its arguments instantiated. There is no such selection mechanism available for reducing the search path in prolog. This implies, that Progres should be more efficient if a large database is involved and some of the arguments are instantiated when the system tries to solve a goal.

There are a number of ways in which the speed might be improved.

On a large database, the user could cause ingres to create secondary indices on the database relations. This should provide some relief but is more of a band-aid than a solution. The ultimate solution would, of course, be the rewriting of ingres so that it provided a pointer to the next tuple for each retrieval. Then only that pointer would have to be stored in the FRAME to ensure correct retrievals during later failures. As an interim measure, `make_ingres()` could be modified so that the target string used in `get_instance()` is only built once and stored with the attribute chain. Then `get_instance()` would be able to copy the string to an appropriate 'C' variable as needed instead of having to construct it each time. This modification should significantly reduce the time it takes `get_instance()` to retrieve a tuple.

6.2 Summary and Conclusions

This thesis has demonstrated that it is possible to interface a deductive interpreter to a relational database system. The particular form that the implementation took demonstrated that there is no impediment to interfacing prolog to any relational database. The interface modifies only the prolog interpreter and uses tools provided by the database system for accessing the database. Furthermore, it seems probable that any database management system that permitted a users application to examine its schema definition (relation, and attribute relations in ingres) could have an interface to prolog built for it.

Examples have shown that use of deductive rules can result in facts being retrieved which are not explicitly defined in the database. There are two implications in this result. The first is that databases can be made smaller because data can be implicitly stored in rules. The second is that the user can automate the use of the rules he normally applies manually to the data in his database allowing him to make his decisions more quickly and probably more accurately.

Writing "user friendly" man/machine interfaces has been demonstrated to be relatively easy in C-Prolog by Warren et al [16] and Neves et al [13]. Having access to a very large database behind such an interface would be very attractive. Ingres directly provides data about the number of tuples which can satisfy a query. Use of this knowledge and creation of secondary indices (another function provided by ingres) could make query optimization relatively straight forward so that even unsophisticated (but difficult) queries entered by a naive user could be optimized without fine tuning the optimization strategy for the particular database or application.

The analysis at the beginning of this chapter has shown that the Progres prototype has many deficiencies. Methods of overcoming the deficiencies are generally fairly obvious though being obvious and being easy are not necessarily the same thing.

Current research in the Japanese fifth generation project and similar projects in Britain and the United States indicate that intelligent, fast access to a large database is likely to be a

necessity in the future. While using C-Prolog and Ingres as the vehicle for this kind of system may not be the most practical or efficient method for achieving this, the Progres system does demonstrate the possibilities and provides directions for further research.

REFERENCES

1. Chomicki, J., and W. Grudzinski "A database support system for PROLOG," in Proceedings of the Logic Programming Workshop '83 26 Jun - 1 July, 1983, pp. 290 - 303, Universidade Nova De Lisboa, Lisbon, 1983.
2. Dahl, V., "On Database Systems Development Through Logic," Transactions On Database Systems, vol. 7 , no. 1, pp. 102 - 123, Association for Computing Machinery, New York, 1982.
3. Demolombe, Robert, "Estimation of the Number of Tuples Satisfying a Query Expressed in Predicate Calculus Language," in Proc of the 6th International Conference on Very Large Data Bases, pp. 55 - 63, IEEE, 1980.
4. Fishman, D. H., "The DBMS - Expert System Connection," New Directions for Database Systems May 16-18, 1984 (Advance Draft), pp. 108 - 120, NYU, New York.

5. Futo, I., F. Darvas, and P. Szeredi, "The Application of Prolog to the Development of QA and DBM Systems," in Logic and Data Bases, ed. Herve Gallaire and Jack Minker, pp. 347 - 376, Plenum Press, New York, 1978.
6. Gallaire, H., "Impacts of Logic on Data Bases," in Proc of the 7th Conference on Very Large Data Bases, pp. 248 - 259, IEEE, 1981.
7. Grant, John and Jack Minker, "Optimization in Deductive and Conventional Relational Database Systems," in Advances in Data Base Theory - Vol 1, ed. Herve Gallaire, Jack Minker, Jean Marie Nicolas, pp. 195 - 234, Plenum Press, New York, 1981.
8. Hammer, M. and S. B. Zdonik, Jr., "Knowledge-Based Query Processing," in Proc of the 6th International Conference of Very Large Data Bases, pp. 137 - 147, IEEE, 1980.
9. Imielinski, T. and W. Lipski, Jr., "On Representing Incomplete Information in a

Relational Database," in Proc of the 7th International Conference on Very Large Data Bases, pp. 388 - 397, IEEE, 1981.

10. Jarke, M., J. Clifford, and Y. Vassilou, "An Optimizing Prolog Front-End to a Relational Query" ACM-SIGMOD 1984, pp. 296-306

11. Kellogg, C. and L. Travis, "Reasoning with Data in a Deductively Augmented Data Management System," in Advances in Data Base Theory - Vol 1, ed. Herve Gallaire, Jack Minker, Jean Marie Nicolas, pp. 261 - 296, Plenum Press, New York, 1981.

12. Maiert, D., "Databases in the Fifth Generation Project: Is Prolog a Database Language?," in The 1984 NYU Symposium (Advanced Draft), "New Directions for Database Systems" May 16 - 18 1984, pp. 82 - 107, NYU, New York.

13. Minker, J., "An Experimental Relational Data Base System Based on Logic (Or Clause Encounters of a Logical Kind)," in Logic and Data Bases, ed. Herve Gallaire and Jack Minker, pp. 107 - 147, Plenum Press, New York, 1978.

14. Neves, J.C., S.O. Anderson, and M.H. Williams, "A Prolog Implementation of Query-By-Example," in Proc of the International Computing Symposium 1983 on Application Systems Development, pp. 318 - 332, B.G. Teubner, Stuttgart, 1983.
15. Parsaye, K., "Database Management, Knowledge Base Management and Expert System Development in PROLOG," in Proceedings of the Logic Programming Workshop '83 26 Jun - 1 July, 1983, pp. 271 - 289, Universidade Nova De Lisboa, Lisbon, 1983.
16. Richards, P., "Evaluation of the size of Query Expressed in Relational Algebra", Proceedings ACM-SIGMOD 1981, pp. 155-163.
17. Stabler, Jr., E.P. and E.W. Elcock, "Knowledge Representation in an Efficient Deductive Inference System," in Proceedings of the Logic Programming Workshop '83 26 Jun - 1 July, 1983, pp. 216 - 228, Universidade Nova de Lisboa, Lisbon, 1983.
18. Warren, D.H.D. and F. Pereira in /usr/local/cprolog/chat.typ.

19. Yokota, H., et al. "An Enhanced Inference Mechanism for Generating Relational Algebra Queries", Proceedings Symposium on Principles of Database Systems 1984, pp. 229-238.

```
/* init : Initial boot file for building the Prolog
system
```

Fernando Pereira
Updated: 27 July 83

```
*/
```

% Atoms that the code requires

```
[].  
{ }.  
end_of_file.  
true.  
user.  
$live.  
$break.  
< .  
= .  
> .  
$yes.  
$no.  
- .
```

% Functors that the code requires

```
call(_).  
,(_,_).  
{ }(_).  
,(_,_).  
,-(_,_).  
,-(_,_).  
$hidden_call(_).
```

% And now we start properly.

% Predicates are attached to evaluable predicates
% defined in the code by one of two methods:

```
%  
% 1) pred(....) :- N.  
% 2) :- $sysp(pred(....),N).  
%
```

% where N is an integer specifying the evaluable
% predicate - N is the value of the switch label in
% "main.c". The difference between these is that with
% the first method a (local) frame will be built which
% will contain the arguments to the procedure; whereas
% in the second case no frame is built.

% Set up the Operators

```

:-op(P,T,A),76).

:-op(1200,fx,[ :- , ?- ]\').
:-op(1200,xfx,[ (:-) , --> ]\').
:-op(1100,xfy,';').
:-op(1050,xfy,'->').
:-op(1000,xfy,'^').
:-op(900,fy,[ \+ , not , spy , nospy ]\').
:-op(700,xfx,[ =, is, =.., ==, \==, @<, @>, @=<, @>=,
              =:=, =\=, <, >, =<, >= ]\').
:-op(500,yfx,[ +, -, /\, \/ ]\').
:-op(500,fx,[ (+), (-), \ ]\').
:-op(400,yfx,[ *, /, //, <<, >> ]\').
:-op(300,xfx,[ mod ]\').
:-op(200,xfy,'^^').

```

~ And now all the evaluable predicates

```

$sysp(F,N) :- 100.
$sysflgs(F,N) :- 101.
:-$sysp((A,B),6).
call(P) :- 1.
:-$sysp($hidden_call(P),121).
:-$sysp(!,2).
true.
repeat :- 3.
:-$sysp(abort,4).
:-$sysp($call(_),5).

$not(X) :- $user_call(X), !, fail.
$not(X).

\+(X) :- $not(X).
not(X) :- $not(X).

(A -> B ; C) :- !, $cond(A,B,C).

$cond(A,B,C) :- A, !, B.
$cond(A,B,C) :- C.

(A;B) :- $call(A).
(A;B) :- $call(B).
(A -> B) :- A, !, B.

see(F) :- 10.
seeing(F) :- 11.
seen :- 12.
tell(F) :- 13.
telling(F) :- 14.

```

```
told :- 15.
close(F) :- 16.
read(T) :- 17.
write(T) :- 18.      display(T) :- 274.      ? + 256
nl :- 19.            ttynl :- 275.
get0(C) :- 20.       ttyget0(C) :- 276.
get(C) :- 21.        ttyget(C) :- 277.
skip(C) :- 22.       ttyskip(C) :- 278.
put(C) :- 23.        ttyput(C) :- 279.
tab(C) :- 24.        ttytab(C) :- 280.
fileerrors :- 25.
nofileerrors :- 26.
rename(F,F1) :- 27.
writeq(T) :- 28.
sh :- 29.
system(S) :- 30.
```

```
print(V) :- var(V), !, write(V).
```

```
print(X) :- portray(X), !.
```

```
print(X) :- write(X).
```

```
% define arithmetic operators
```

```
% nullary ops
```

```
:-$sysflgs(cputime,1).
:-$sysflgs(heapused,2).
```

```
% unary operations
```

```
:-$sysflgs([A|B],1).
:-$sysflgs(+(A),1).
:-$sysflgs(-(A),2).
:-$sysflgs\(A),3).
:-$sysflgs(exp(A),4).
:-$sysflgs(log(A),5).
:-$sysflgs(log10(A),6).
:-$sysflgs(sqrt(A),7).
:-$sysflgs(sin(A),8).
:-$sysflgs(cos(A),9).
:-$sysflgs(tan(A),10).
:-$sysflgs(asin(A),11).
:-$sysflgs(acos(A),12).
:-$sysflgs(atan(A),13).
:-$sysflgs(floor(A),14).
```

```
% binary operations
```

:-Ssysflgs(A+B,1).
:-Ssysflgs(A-B,2).
:-Ssysflgs(A*B,3).
:-Ssysflgs(A/B,4).
:-Ssysflgs(A mod B,5).
:-Ssysflgs(A \ B,6).
:-Ssysflgs(A \ B,7).
:-Ssysflgs(A<<B,8).
:-Ssysflgs(A>>B,9).
:-Ssysflgs(A/B,10).
:-Ssysflgs(A^B,11).

?: Unary predicates

^S+(A,X) :- 131.
^S-(A,X) :- 132.
^S(A,X) :- 133.
Sexp(A,X) :- 134.
Slog(A,X) :- 135.
Slog10(A,X) :- 136.
Ssqrt(A,X) :- 137.
Ssin(A,X) :- 138.
Scos(A,X) :- 139.
tan(A,X) :- 140.
Sasin(A,X) :- 141.
Sacos(A,X) :- 142.
Satan(A,X) :- 143.
Sfloor(A,X) :- 144.

?: Binary operations

^S+(A,B,X) :- 151.
^S-(A,B,X) :- 152.
^S*(A,B,X) :- 153.
^S/(A,B,X) :- 154.
Smod(A,B,X) :- 155.
^S/(A,B,X) :- 156.
^S/(A,B,X) :- 157.
^S<<(A,B,X) :- 158.
^S>>(A,B,X) :- 159.
^S/(A,B,X) :- 160.
^S^(A,B,X) :- 161.

X is Y :- 40.
X == Y :- 41.
X \= Y :- 42.
X < Y :- 43.
X > Y :- 44.
X =< Y :- 45.

X >= Y :- 46.

var(X) :- 50.

nonvar(X) :- 51.

integer(X) :- 52.

number(X) :- 91.

atomic(X) :- 53.

atom(X) :- 59.

primitive(X) :- 118.

db_reference(X) :- 119.

statistics :- 93.

prompt(,_ _) :- 31.

exists(user) :- !.

exists(_) :- 32.

\$save(F,I) :- 33.

save(F) :- \$save(F,_).

save(F,I) :- \$save(F,I).

X=X.

functor(T,F,N) :- 56.

arg(N,T,A) :- 57.

X=..L :- 58.

name(X,L) :- 75.

A == B :- 54.

A \== B :- 55.

A @< B :- 86.

A @> B :- 87.

A @=< B :- 88.

A @>= B :- 89.

clause(P,Q) :- !, \$checkkey(P),
 \$clause((P:-Q),R,P).

clause(P,Q,R) :- var(R), !, \$checkkey(P),
 \$clause((P:-Q),R,P).

clause(P,Q,R) :- instance(R,(P:-Q)).

assert(C) :- 61. assertz(C) :- 61.

asserta(C) :- 62.

assert(C,R) :- 63. assertz(C,R) :- 63.

asserta(C,R) :- 64.

\$clause(_,_) :- 65. \$clause(_,_):-66.

retract(C) :- \$retract(C).

\$retract(V) :- var(V), !, fail.


```
Sread_file(F,S) :-
    seeing(I),
    telling(O),
    see(F),
    Scycle(S),
    close(F),
    see(I),
    tell(O),
    fail.
```

```
Sread_file(,_).
```

```
Scheckfile(F) :-
    (atom(F),
     ttyNl, display('Invalid file name: ',
                    display(F), ttyNl, fail
```

```
    exists F,
     ttyNl, display('The file ', display(F),
                    display(' does not exist.', ttyNl, fail
```

```
* break *
```

```
break :- Sbreak Sbreak.
```

```
Sbreak :-
    Sbreak_start,
    Sread_file(user,O),
    Sbreak_end,
    Sexit_break.
```

```
Sbreak :- Sexit_break. % just to make sure
```

```
expand_exprs X,Y :-
    Scaith(XO,XO),
    Sflag_code(XO,XO),
    nonvar(Y),
    Sflag_code(YO,YO),
    Scaith(,YO).
```

```
Scaith(X,Y) :- !0.
```

'NOLC' :- 70.
'LC' :- 71.

% Various hacky system predicates

```
:-$sysp($break(_),102).
:-$sysp($exit_break,103).
$prompt(_):-104.
$user_exec(_):-105.
:-$sysp($save_read_vars,106).
:-$sysp($reset_read_vars,107).
:-$sysp($reply,108).
$recons(_):-109.
:-$sysp($break_start,110).
:-$sysp($break_end,111).
$assertr(_):-112.
:-$sysp($rest_in_peace,113).
$user_call(_):-114.
$repeat :- 3.
$read(X) :- 17.
```

% How to get out (various mnemonics)

```
:- $sysp(halt,113).
```

% Interpreter control

\$live:- \$cycle(0), \$rest_in_peace.
% this is the Prolog session goal

```
$cycle(0):-
    prompt(Old, ' '),
    $repeat,
    $prompt(' ?- '), $read(C),
    $interpret(C,0),
    prompt(_ Old),
    !.
```

```
$cycle(Status):-
    prompt(Old, ' '),
    $repeat,
    $read(C),
    $interpret(C,Status),
    prompt(_ Old),
    !.
```

```
Sinterpret(C,Status) :-
    var(C),
    display('! Statement is a variable',
    ttynl,
    fail.
```

```
Sinterpret(end_of_file,_):-!
```

```
Sinterpret(C,Status):-
    Sdirective(C,Status,D,T),
    prompt:Old,
    Sdogoal(T,D),
    prompt(_,Old),
    fail.
```

```
Sinterpret(C,Status):-
    expand_term(D,C),
    Sassert(C),
    fail.
```

```
Sassert(C):-Sassert(C),
Sassert(C):-
    seeing('user'),
```

```
Sassert(C):-
    display(' clause: ',
    display(C),
```

```
Sdirective :- (X,_,X,command):-!,
Sdirective :- (X,_,X,question):-!,
Sdirective(X,0,X,question).
```

```
Sdogoal(command,C):-
    Suser_exec(C),
```

```
Sdogoal(command):-
    ttynl, display('?',
    ttynl.
```

```
Sdgoal(question,Q) :-
    $save_read_vars,
    $user_exec(Q),
    $reply,
    ttyln, display(yes),
    $reset_read_vars,
    ttyln,
```

```
Sdgoal(question,_) :-
    ttyln, display(no),
    $reset_read_vars,
    ttyln,
```

* consult and reconsult *

```
consult(X) :- $break($cresult(0,X)).
```

```
reconsult(X) :- $break($cresult(1,X)).
```

```
[X|Rest] :- $conlist([X|Rest]).
```

```
$conlist([]) :- !.
```

```
$conlist([_X|Rest]) :- !, reconsult(X), $conlist(Rest).
```

```
$conlist([X|Rest]) :- !, consult(X), $conlist(Rest).
```

```
$cresult(R,F) :-
```

```
    $0 is heapused,
    $T0 is cputime,
    $recons(R),
    $checkfile(F),
    $readfile(F,consult),
    $Tt is cputime-$T0,
    $Ts is heapused-$0,
    display(F), $dpr(R),
    display(Ts), display(' bytes '),
    display(Tt), display(' sec. '), ttyln,
    fail.
```

```
$cresult(R,F) :- $exit_break.
```

```
$dpr(0) :- !, display(' consulted '),
```

```
$dpr(1) :- display(' reconsulted '),
```

```
Sread_file(F,S) :-
    seeing(I),
    telling(O),
    see(F),
    Scycle(S),
    close(F),
    see(I),
    tell(O),
    fail.
```

```
Sread_file(,_).
```

```
Scheckfile(F) :-
    (atom(F),
     ttynt, display('Invalid file name: '),
     display(F), ttynt, fail
    ),
    (exists(F),
     ttynt, display('The file '), display(F),
     display(' does not exist.'), ttynt, fail
    ),
    !.
```

```
/* break */
```

```
break :- Sbreak(Sbreak).
```

```
Sbreak :-
    Sbreak_start,
    Sread_file(user,O),
    Sbreak_end,
    Sexit_break.
```

```
Sbreak :- Sexit_break. % just to make sure
```

```
expand_exprs(X,Y) :-
    Scarith(XO,XO),
    Sflag_code(XO,X),
    nonvar(Y),
    Sflag_code(YO,Y),
    Scarith(,_YO).
```

```
Scarith(X,Y) :- 170.
```

```
all_float(X,Y) :-  
    all_float(X0,X0),  
    $flag_code(X0,X),  
    nonvar(Y),  
    $flag_code(Y0,Y),  
    Sall_float(_,Y0).
```

```
Sall_float(X,Y) :- 90.
```

```
$flag_code(1,on).  
$flag_code(0,off).
```

```
expand_term(T0,T) :-  
    term_expansion(T0,T1), !,  
    Sexpand_term(T1,T).  
expand_term(T0,T) :- Sexpand_term(T0,T).
```

```
Sexpand_term(T0,T) :- Stranslate_rule(T0,T), !.  
Sexpand_term((H:-B0),(H:-B)) :-  
    Scompile_arith(B0,B), !.
```

```
Sexpand_term(T,T).
```

```
end_of_file.
```

% needed here for a special reason...

END

1 8 0 4 8 6

FIN