1

MicroD

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform 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.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme 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.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, S.R.C. 1970, c. C-30, et ses amendements subséquents.

Prolog in 'C'

by

Joanne Lee Boyd, B.Eng.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

March 1, 1989

© Joanne Lee Boyd 1989

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-51188-5

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

Prolog in 'C'

submitted by Joanne Lee Boyd, B.Eng., in partial fulfilment of the requirements for the degree of Master of Engineering.

Chairman,
Department of Systems and
Computer Engineering

Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
April 12, 1989

The result is not only readable, but can be *tuned* by the programmer to improve the performance. The translator output, which will take advantage of several known optimization techniques, is not expected to run more efficiently than the input would with a proven interpreter. Therefore tuning, which might consist of simply removing redundant statements, useless looping, or taking advantage of argument modes, can improve the programs execution. The extreme case to tuning is *hand-compiling*; i.e., a programmer who is confident in his understanding of the PIC definition, can by-pass the translation phase and write PIC procedures directly. All built-in logic functions supplied with the *Prolog in 'C'* development environment are hand-compiled PIC code: they represent very efficient versions of their operations.

interfacing — The third objective is to provide a simple interface between the two programming techniques. The ability to produce hand-compiled PIC establishes a method for writting 'C' functions which can be invoked by Prolog functions. A hand-compiled PIC procedure which simply follows and accepts the standard calling sequence but executes strictly sequential 'C' code, represents a 'C' function which can be called by Prolog. If 'C' and Prolog can be mixed in a hand-compiled PIC procedure, then a flexible link between the two languages has been reached.

1.3 Thesis Overview

Originally Prolog execution was performed by an interpreter. This inherently makes it more difficult to see the procedural aspects of a program. However, given any program and its interpreter, it is always possible to produce a set of sequential instructions. In the worst case this is defined by a step-by-step imitation of the interpretation [23]. D.H.D Warren introduced an abstract instruction set, WAM⁴, in which a Prolog program can be redefined and optimized. His work lead to the design of a designated Prolog machine (architecture) which was very efficient but impractical (costly) [28]. The importance and permanence of the von Neumann

⁴Warren's Abstract Machine

Acknowledgements

I would like to thank my supervisor, Dr. Gerald M. Karam, for his attention and never ending enthusiasm.

I would also like to thank my family and friends for their doubtless faith and encouragement.

Financial support was provided by NSERC, and Computing Devices Company.

Contents

1	Introduction	1
1.1	Generations of Software Agree	1
1.2	Thesis Objectives	3
1.3	Thesis Overview	4
2	Background	6
2.1	The Prolog Machine – Description	6
2.1.1	The Program	7
2.1.2	The Model	8
2.1.3	The Matching Algorithm — Unification	12
2.1.4	Special Features of Prolog	16
2.2	The Prolog Machine – Implementation	18
2.2.1	Clause Representation	18
2.2.2	Data Representation	20
2.3	An Interpreter Architecture	29
2.3.1	Arguments	31
2.3.2	The Stacks	31
2.3.3	The Clause Space	35
2.3.4	The Atom Table	35
2.3.5	Effects of Execution	35
2.4	Warren’s Abstract Machine - WAM	36
2.4.1	Data Representation and Classification	36
2.4.2	Instruction Set	39
2.4.3	Instruction Optimizations	44
2.5	Concluding Remarks	44
3	General Design of <i>Prolog</i> in ‘C’	45
3.1	Requirements	47
3.2	The Basic PIC Control Strategy	52
3.3	Features of PIC Clauses	61
3.3.1	A Simple Clause	62

3.3.2	Clause Determinancy	65
3.3.3	The or Predicate	69
3.3.4	Dynamic Clauses	76
3.3.5	Meta-evaluation and Negative Logic	79
3.3.6	Tail Recursion Optimization	80
4	Unification in PIC	82
4.1	'C' -Prolog Interface	82
4.2	Data Representation	89
4.2.1	Data Typing	89
4.2.2	Variable Classifications	90
4.2.3	Data Storage	92
4.3	Unification	93
4.3.1	Dereferencing	94
4.3.2	Binding	94
4.3.3	Trailing	96
4.4	Example	97
5	Optimizations	98
5.1	Tuning the PICG Output	98
5.1.1	Simple Adjustments	99
5.1.2	Type and Mode Optimizations	99
5.1.3	Optimization Of Determinate Procedures	103
5.2	'C' Functions in a Prolog Environment	107
5.3	Hand-compiling PIC	109
5.3.1	Variable Access	110
5.3.2	Backtrackable Hand-compiled PIC	112
6	PIC and PICG Analysis	118
6.1	Benchmark Test	119
6.2	Exam Scheduler	121
6.3	Interpreter Test	123
6.4	Large Prolog Project	124
7	Conclusions	128
A	Supplementary Background Information	135
A.1	Interpretation of an Example Query	135
A.2	Data Representation Examples	138
A.2.1	Structure Sharing	138
A.2.2	Structure Copying	141
A.3	Comparison of Structure Sharing and Structure Copying	143

A.4	Effects of Execution on the Components of an Interpreter	145
A.5	An Optimizing Feature of Prolog	147
A.6	WAM Unification Instructions	148
A.6.1	Get Instructions	148
A.6.2	Unify Instructions	149
A.6.3	Put Instructions	151
A.7	WAM Control Instructions	152
A.8	WAM Indexing Instructions	155
A.9	WAM Indexing Instructions -- Example	155
B	Selected 'C' Support Software	159
B.1	manage_action	159
B.2	Differences Between WAM and PIC Unification Instructions	161
C	Detailed Execution Example	163
C.1	Example 1	163
D	Built In Predicates	174
D.1	Control Built-in Predicates	174
D.2	Arithmetic Built-in Predicates	175
D.3	Dynamic Operations	176
D.4	Input/Output Built-in Predicates	177
D.5	Testing Predicates	178
D.6	Character Manipulation	180
E	Testing	181
E.1	Using the PIC Generator	181
E.2	Benchmark Test	182
E.2.1	Append Optimizations	184
E.2.2	Moving Addition in Line of for/3	189
E.3	Exam Scheduler	191
E.3.1	Optimization of all_avail_in_slot/2	196
E.4	Interpreter Test	198
E.4.1	Optimized Version of built_in/1	203

List of Figures

2.1	Simple example program.	9
2.2	Graph representation of the query <code>child(cathy,A)</code>	10
2.3	Pseudo code for a simple Prolog interpreter.	13
2.4	Graph representation for a clause.	19
2.5	A list of clauses in a procedure referenced through the atom table. The procedure in this example contains two clause alternatives. . . .	21
2.6	Value cell for the structure — <code>blue(X,mix([X a],b))</code>	23
2.7	Illustration of two variables which <i>dereference</i> to <code><atom,cathy></code> . . .	25
2.8	Prototype for the structure — <code>t(p(X),q(p(X),Y))</code>	28
2.9	Data objects and pointers in an interpreter implementation.	32
2.10	<code>qsort/3</code> predicate.	38
3.1	Prolog and 'C' Multi-language Development Environment.	46
3.2	Mode switching allowed in the target program.	49
3.3	Pseudo-code for the basic PIC procedure definition.	53
3.4	Example program for illustrating control flow.	54
3.5	Flow of procedure calls using a typical imperative approach.	55
3.6	Illustration of a continuation list of function closures.	56
3.7	Flow of procedure calls using continuation passing.	58
3.8	Flow of procedure calls using continuation passing and determinate clause returns.	60
3.9	Functionality of <code>manage_action</code> for a simple clause.	64
3.10	Description of the control instructions for a cut clause.	67
3.11	Pseudo-code to illustrate clause indexing - Clause 1.	70
3.12	Pseudo-code to illustrate clause indexing - Clause 2.	71
3.13	Pseudo-code to illustrate clause indexing - Control Code.	72
3.14	Example of a continuation containing <code>or</code> calls.	75
4.1	<code>member/2</code> predicate	84
4.2	The PIC value cell <i>mini stack</i> for <code><integer,3></code>	89
4.3	<code>qsort/3</code> predicate.	90

4.4	Dereferencing Algorithm	95
5.1	Quick indexing scheme for member(+,+)	102
5.2	count/2 predicate as it is typically written.	103
5.3	A strictly determinate version of the count/2 predicate.	104
5.4	count/2 determinate procedure in optimized 'C' form.	106
5.5	PIC version of n1/0 built-in predicate.	108
5.6	Optimized PIC version of the n1/1 predicate.	111
5.7	Hand-compiled version of the unify/2,=/2 , predicate.	113
5.8	Optimized version of the hand-compiled PIC function repeat/0	115
5.9	Optimized version of the hand-compiled read_char/1 predicate.	116
6.1	function_lookup/2 predicate.	126
A.1	Prototype #1 developed for equation (2)	139
A.2	Prototype #2 developed for equation (3)	139
A.3	Prototype #3 developed for equation (5)	139
A.4	Prototype #4 developed for equation (6)	140
A.5	member/2 predicate.	148
A.6	WAM instruction format for a fact.	153
A.7	WAM instruction format for a multiple subgoal rule.	154
A.8	WAM instruction format for a single subgoal rule.	154
A.9	concatenate/3 predicate.	156
A.10	Example of WAM compiler output for the <i>concatenate</i> procedure.	157
C.1	Function declaration and local variables of the member2 PIC function.	164
C.2	First clause of the member2 PIC function.	165
C.3	Second clause of the member2 PIC function.	166
C.4	Switch block of the member2 PIC function to control execution according to the next_action status returned by manage_action	167
C.5	Goto labels of the member2 PIC function.	168
C.6	Argument mini stack and copy stack passed in the 'C'- member2 call: member2(3,[1,3,5])	170
C.7	Argument mini stack after the get instructions of the first call are completed.	172
C.8	Argument mini stack to be passed to the first subgoal of the second clause. TRO is not being considered yet.	173
E.1	Optimized version of built_in/1	204
E.2	Minimum mainline and CPI requirements for nrev/0	205

List of Tables

2.1	Definitions of terms used in interpreter pseudo-code.	14
2.2	Arguments passed to gold/5 by green/1 in a structure sharing implementation.	27
2.3	Arguments passed to gold/5 by green/1 in a structure copying implementation.	30
6.1	Results of the benchmark test.	120
6.2	Execution times (sec) for the Exam Scheduler.	122
6.3	Maximum memory requirements for the Exam Scheduler.	122
6.4	Execution times (sec) for the ODA Analyzer.	127
B.1	Differences between WAM and PIC get , put , and unify instructions	162

Chapter 1

Introduction

1.1 Generations of Software Agree

Can the generations work in harmony? It seems to be worth the effort when referring to third¹, fourth², and fifth generation³ programming languages. It is certainly the topic of many papers [19, 18, 32, 24], a selling point for several products [6, 27, 8], and the principle motivation for this thesis. Arguments which support the combination of logic and procedural programming are of particular interest, but lead to the question of how; that is, what will it take to close the “generation gap”.

An examination of the strengths and weaknesses of each technique is the standard approach to arguing their compatibility, thus answering the question: Why bother? Logic programming boasts a “natural” tendency to rapidly produce clear, readable and concise programs with few errors [31]: this is a consequence of its simple and clear syntax and semantics [11, 26]. Interpreters offer incremental programming of immediately executable and testable code, and the development environments of most logic languages include at least a debugging interpreter. Prolog has been successfully applied to various complex applications, for example, natural language systems, symbolic manipulation, compiler writing, mapping and routing, planning,

¹BASIC, Pascal, and ‘C’ are examples of third generation programming languages

²DBASE III is a fourth generation language.

³Prolog is a fifth generation language.

game playing, and expert systems [26]. In general, problems requiring deductive reasoning of data are well suited to Prolog solutions because of the ability to represent knowledge symbolically. As with any language, there are also weaknesses. Logic programming is not normally chosen for procedures with intense algorithmic sequences or external communications, and is often considered inferior in terms of execution speed [19, 17]. Procedural languages are apparently quite complimentary. They have the advantage of maturity; that is, there is existing expertise associated with these techniques. This is particularly evident in reusable applications such as peripheral drivers and standard math functions. Both these examples were described earlier as problem areas for declarative representation [19, 17]. Speed is also considered to be a strength of procedural languages which are better suited to the standard von Neumann architecture [19]. Their weakness, however, lies in the complex development cycle required to produce readable, maintainable code [19].

Combining logic programs with procedural programs in a multi-language programming environment creates a potentially powerful and flexible method of software development. Several versions of Prolog have acknowledged this and are, therefore, providing external, imperative language interfaces [6, 5, 8, 3]. Researchers have also looked into allowing procedural languages to call background interpreters when a Prolog code segment is to be executed [19, 24]. But the most dedicated believers of merging these techniques are producing new languages which combine imperative style, (e.g. if-then-else and case structures), with logic deduction [27, 17]. The influence behind this thesis comes from yet another approach; namely, the arguments which support the extraction of the procedural aspects of Prolog in order to have efficient declarative programs running on machines which support other languages. This has been exploited previously by the developers of POPLOG [18], however, in this thesis consideration is also given to the importance of a *common* single source target. Having a Prolog and 'C' dual development environment which generates a strictly 'C' system, not only covers all of the combinations discussed, but offers extra portability because, for the most part, existing machines already have supporting 'C' compilers [32]. Thus the goal of this thesis is to define a conventional control structure for Prolog which supports a relatively complete Prolog description, and

to provide a Prolog—‘C’ translator which produces logic functions that are easily integrated into ‘C’ programs.

1.2 Thesis Objectives

A usable, multi-language, single target programming environment is only one objective. It is partially satisfied by the definition of PIC, *Prolog in ‘C’*, and the implementation of a translator to convert Prolog to ‘C’. The complete development environment will include: (1) Prolog code segments, (2) ‘C’ code segments, (3) the Prolog—‘C’ translator, (4) PIC (the translator output), and (5) the necessary interface functions and their generators. This package is expected to be used for the following scenarios.

1. To develop ‘C’ systems which include program segments that are well suited to declarative programming (e.g. a scheduler).
2. As a first step to a ‘C’ implementation of a declarative program. The translator output represents a workable ‘C’ implementation which can be optimized or eventually replaced by a corresponding ‘C’ program. As an example, several solution models might be under consideration. Each case could be rapidly prototyped in Prolog and tested. The chosen solution could then be translated to PIC, thus a ‘C’ implementation is immediately available.

Having a common language target system allows the objective to be extended. If PIC is presented in a consistent and simple enough manner that its execution is understandable to the user, then tuning and hand-compiling can be used to improve the program’s performance. This, in turn, should be used to allow unrestricted switching from declarative to procedural code segments and vice versa. There are, therefore, three main objectives:

- translation* — The objective is to produce a translator which outputs a PIC version of the input Prolog program.
- tuning* — Here, the objective is to produce procedural Prolog in a *common* imperative language through a consistent and well defined compilation procedure.

The result is not only readable, but can be *tuned* by the programmer to improve the performance. The translator output, which will take advantage of several known optimization techniques, is not expected to run more efficiently than the input would with a proven interpreter. Therefore tuning, which might consist of simply removing redundant statements, useless looping, or taking advantage of argument modes, can improve the programs execution. The extreme case to tuning is *hand-compiling*; i.e., a programmer who is confident in his understanding of the PIC definition, can by-pass the translation phase and write PIC procedures directly. All built-in logic functions supplied with the *Prolog in 'C'* development environment are hand-compiled PIC code: they represent very efficient versions of their operations.

interfacing — The third objective is to provide a simple interface between the two programming techniques. The ability to produce hand-compiled PIC establishes a method for writting 'C' functions which can be invoked by Prolog functions. A hand-compiled PIC procedure which simply follows and accepts the standard calling sequence but executes strictly sequential 'C' code, represents a 'C' function which can be called by Prolog. If 'C' and Prolog can be mixed in a hand-compiled PIC procedure, then a flexible link between the two languages has been reached.

1.3 Thesis Overview

Originally Prolog execution was performed by an interpreter. This inherently makes it more difficult to see the procedural aspects of a program. However, given any program and its interpreter, it is always possible to produce a set of sequential instructions. In the worst case this is defined by a step-by-step imitation of the interpretation [23]. D.H.D Warren introduced an abstract instruction set, WAM⁴, in which a Prolog program can be redefined and optimized. His work lead to the design of a designated Prolog machine (architecture) which was very efficient but impractical (costly) [28]. The importance and permanence of the von Neumann

⁴Warren's Abstract Machine

machine was recognized by Mellish who, without ignoring Warren's work, described a new language POPLOG, which is a procedural definition of Prolog [17]. It is a combination of these ideas which motivates the definition of PIC and the design of the Prolog—'C' translator. The background material required to understand the thesis work is quite extensive. This information is given in Chapter II in the following format: (1) a description of the abstract Prolog machine ⁵, (2) the implementation of an interpreter for this model on a von Neumann architecture, and (3) an explanation of the extensions required for WAM-based compiler implementations.

Chapter III reviews the thesis objectives and develops a set of requirements for PIC from which the basic format of a PIC procedure is obtained, and the procedural definition of *Prolog in 'C'* is established. Chapter III develops the control strategy and the basic layout of the PIC code, and Chapter IV explains the details of unification. These features are largely affected by the requirements and definition of the 'C'—Prolog interface which is described in Chapter IV but not implemented. Chapter V describes the approach to tuning and hand-compiling.

In Chapter VI, an evaluation of the translator and its output is presented in terms of coverage, efficiency, and the restrictions imposed by the design. Suggestions and references for improving and extending the implementation are provided as they appear relevant. Finally, the conclusions of Chapter VII consist of a comparison of the results with the objectives in a critical assessment.

⁵Edinburgh syntax is assumed throughout the thesis.

Chapter 2

Background

Compilation and interpretation are the two most common approaches to program execution. The definition of an interpreter can always be used to produce a compilable form of a particular program. In the most extreme case, the code to be compiled is a set of instructions modeled after the interpreter's execution [23]. The history of Prolog follows a similar scenario; that is, Prolog was originally interpreted [26], but is now generally compiled into instructions which access similar data objects to those used in the initial implementations. Many of these newer versions, however, still include an interpreter to perform specific, hard to predetermine tasks [19, 3]. ALS Prolog is a notable exception in that it compiles dynamic predicates¹ [3]. The background required for this thesis is presented in this chapter and consists of: (1) a description of an interpreter, (2) an implementation on a von Neumann machine, and finally (3) an explanation of the WAM-based compilers of which PIC is a member. Each section introduces terminology and concepts necessary to the understanding of subsequent discussions.

2.1 The Prolog Machine – Description

High-level programming languages, like Prolog, attempt to ease the task of the programmer by providing a hardware independent abstract model which can be

¹These are statements which are asserted and retracted during the program's execution.

compiled or interpreted into machine specific code for execution on the target architecture. The language models consist of a syntax and a set of related semantic rules which define the requirements for their execution, and thus their implementation. Prolog is based on predicate logic and resolution theorem proving. A program consists of a series of logic statements called *facts* and *rules*, and the deductive model is a limited and controlled version of *backward chaining* [21]. This section describes the Prolog machine at an abstract level in order to present the framework of the implementation.

2.1.1 The Program

A Prolog program consists of a set of logic statements which define directly or through deduction, the states, and relationships between, the objects being considered. There are two formats for these statements: *facts* and *rules*.

Facts

A fact defines a **true** term state, (object state), or a **true** relationship between objects. For example,

```
mother(betty,john).
```

is a fact which states that: betty is the mother of john. The predicate symbol (*functor*) of this example is **mother**. A *predicate* defines an object state or a relationship between objects. For example **mother** is the relation holding **betty** and **john** together, and in **red(hat)**, **red** defines the state of **hat**. However, to completely define a *predicate*, the number of arguments (*arity*) must be provided along with the *functor*. Predicates are represented by a *functor/arity* pair; for example **mother/2** and **red/1**. The symbols **betty**, **john** and **hat** are *arguments (terms)* of the fact. Therefore, a *predicate* is the general definition of a state or relationship, and a *fact* is a specific and true representation of the predicate.

Rules

A rule is a statement which claims that a relationship (a *goal*) holds true if a conjunction of one or more other goals can be proven to hold true. For example, a family relationship rule is:

```
child(X,Y) :- mother(Y,X).
```

This rule states that *X* is a child of *Y*, if *Y* is the mother of *X*. The goal `mother(Y,X)` must be proven in order to infer that `child(X,Y)` is true. The term *goal* implies that a proof is required to infer any truth about the relation.

A rule consists of two parts: (1) a *head goal*, and (2) a conjunction of *body goals* called *subgoals*. In the example, `child(X,Y)` is the head goal, and `mother(Y,X)` is a single subgoal. The predicate of a rule is defined by the head goal; e.g. `child/2`. The arguments of this rule are predicate *variables* which are syntactically represented by symbols beginning with an upper-case letter or an underscore. The arguments of a rule are much like the local variables of a Pascal procedure; that is, they take on the values passed in a call, and their identifier is strictly local to the statement definition unless passed in a successive call.

Example

Figure 2.1 provides a simple program example which contains the two examples discussed above. Collections of program *clauses* (statements) with the same predicate are called *procedures*². There are three procedures in this example: `sibling/2`, `mother/2`, and `child/2`.

2.1.2 The Model

A program is executed by posing a *query*, which is a user supplied goal; i.e a user introduced relation which is to be proven through deduction using the program statements. In a backwards control strategy, the proof starts with the query goal. The complete proof is equivalent to a graph whose root node is the query goal.

²Several references use *predicate* synonymously with *procedure*.

<code>sibling(cathy, john).</code>	fact #1
<code>mother(betty, john).</code>	fact #2
<code>child(X,Y) : - mother(Y,X).</code>	rule #1
<code>child(X,Y) : - sibling(X,Z), child(Z,Y).</code>	rule #2
$\underbrace{\text{sibling}(X,Y)}_{\text{HEAD}} : - \underbrace{\text{sibling}(Y,X)}_{\text{BODY}}.$	rule #3

Figure 2.1: Simple example program.

The remaining nodes represent the subgoals introduced by rules used during the execution, and are reached by matching the source node's goal predicate with the program clause predicates. An edge of the graph represents the fact or rule used to continue through that path of the proof, and there are as many paths leaving a node as there are clauses in the matching procedure. A fact, which does not introduce any new goals, will terminate each path of the proof.

For a particular query, the proof moves along the paths of the graph by making the most general substitution which makes the edge goal, (the rule head goal or fact), equivalent to the node goal. The destination node must now reflect this substitution, and therefore the proof is specific to the query. This substitution process is called *unification*. When a proof fails to match the goals, an alternate path is taken. If all paths fail, the proof backtracks to the previous (*parent*) node and tries one of its alternative paths. For example, the query `child(cathy,A)`, which asks if `cathy` can be proven to be someone's child, would expand into the proof tree given in Figure 2.2 when the program in Figure 2.1 is used. The diagram shows the different paths (*alternatives*) which can be taken from each node. The variables of a clause are much like the local variables of a Pascal procedure, and are therefore replaced with unique names for each call to avoid confusion. These are shown as capitalized letters preceded by an underscore; e.g. `_B` has replaced the `Z` of rule #2.

Figure 2.2: Graph representation of the query `child(cathy,A)`.

A depth-first proof strategy proceeds along each branch of the proof graph until a termination point is reached before exploring another path. Prolog is a depth-first, top-down, left-to-right proof strategy. This implies that clauses are tried as they appear in the program from the top to the bottom of the definition, and subgoals are proven completely left-most first. The path numbers on the edges of Figure 2.2 indicate the order in which branches are explored by a Prolog interpreter.

In the example of Figure 2.2, rule #1 is used to start the proof and the traversal of path 1. The rule introduces a new node defined by the body of the clause which contains the goal `mother(A,cathy)`. During the search, the most general unifier required to make a node goal match a fact or rule's head goal is applied across the clause before a success is declared or a new node is created respectively. In path 1, the local variable `X` is replaced by `cathy`, and `Y` is replaced by `A`. This path ends at this node because, although fact #2 has a matching predicate to the new node goal, `mother(A,cathy)` cannot be unified with `mother(betty,john)`. This branch of the proof tree *fails* to prove the query the goal, therefore the search backtracks to the previous node, which in this case is the query node. A different `mother/2` query goal, for example `mother(betty,F)`, would be represented by a graph equivalent to that of Figure 2.2 with respect to the edges (clauses) taken. The arguments and substitutions would mark the differences. This second example query would have *succeeded*, (been proven), by taking path 1.

When path 1 fails to prove `child(cathy,A)`, the traversal of path 2 begins. The first edge of this path is formed by rule #2 which follows to a node consisting of the two subgoals as shown. The requirement to prove the left-most subgoal first causes fact #1 to be matched to the first subgoal. This provides the substitution `_B = john`, and creates a new node containing the remaining subgoals as they would appear after unification. This path continues with the proof of the newest node goal which is matched with fact #2. The proof terminates successfully and unifies the variable `A` with `betty`. The path marked 3 is another potential route in the solution of this query. The graph is infinite along this path because rule #3 is left-recursive, however, alternative solutions are not explored unless requested by the user.

A common approach to depth-first graph searches used in computer program-

ming is to build a stack of nodes visited. In this case, a procedure definition provides the adjacent nodes; i.e. each clause with a matching predicate to a given node goal is *adjacent* to, (creates an edge from), that node. The ordering of clauses in the procedure “adjacency list” is top-down with respect to the program definition. When a node is created, it is pushed onto the stack of visited nodes such that the left-most goal is on the top of the stack. The stack grows as long as rules are required to prove a node. When a node cannot be proven, the execution *backtracks* to the previous node and tries the next alternative. The query is successfully proven if the search reaches a termination node. Figure 2.3 presents a pseudo-code description of the depth first search used in Prolog interpretation, and Table 2.1 supplies the definitions of the terms used in this code.

The resolution stack and execution stack introduced in the code are implementation details which are revealed by this algorithmic representation of the interpreter. An example query is given in § A.1 which uses this description to further illustrate the deduction of a proof. The example reveals the implementation requirements of a simple Prolog interpreter.

2.1.3 The Matching Algorithm — Unification

In the previous section, *unification* was defined as the act of producing the most general substitution required to make two goals identical. The interpreter scanned the program until a matching predicate was found, and then unified the goal with the fact or rule’s head goal. This section will explain in detail the unification algorithm.

The unification algorithm used in Prolog is similar to that of logic. Both check if two entities can be matched, and make the minimum substitution required for their equivalence. Unification is performed *term by term*, (argument by argument) A *term* is defined as a constant, a variable or a structure, where a structure is a compound term with its own functor/arity³. A clause or term which does not include any variable terms is said to be *ground*. The algorithm is defined as follows:

- Two terms unify successfully if they are identical constants.

³Some references will use *structure* to describe a zero arity constant term like `john` as well as compound terms.

```

WHILE there are goals to prove DO
    Try to prove the goal at the top of the resolution stack.
    Find the program clause(s) with the same predicate
    as the current goal.
    IF (no matching clause is found) THEN
        IF (a choice point has been marked) THEN
            Backtrack.
        ELSE Leave the loop and indicate failure.
        ENDIF
    ELSE
        IF (more than one clause has the same predicate) THEN
            Indicate that a choice point has been reached.
        ENDIF
        Call the first matching clause.
        Unify the goal and the clause.
        IF (the unification was unsuccessful) THEN
            IF (a choice point has been marked) THEN
                Backtrack.
            ELSE Leave the loop and indicate failure.
            ENDIF
        ELSE Push the subgoals of the matching clause onto the
            resolution stack.
        ENDIF
    ENDIF
ENDIF
ENDWHILE

```

Figure 2.3: Pseudo code for a simple Prolog interpreter.

<i>parent</i>	The rule from which a subgoal was introduced.
<i>clause</i>	A query, fact, or rule.
<i>current goal</i>	The goal which is the current focus during execution. The initial current goal is the query goal.
<i>choice point</i>	A point during execution where there is more than one unexplored clause with a predicate that matches the current goal.
<i>alternatives</i>	Clauses which match the current goal which have not yet been used in the execution of the proof; the clauses of a choice point.
<i>nearest choice point</i>	The most recently acknowledged choice point.
<i>resolution stack</i>	A LIFO stack of goals yet to be proven.
<i>execution stack</i>	Analogous to a Pascal call stack.
<i>instance</i>	A copy of a clause where all variables have been renamed. When a clause is called, a local representation is built on the execution stack in a manner analogous to that of an activation record of a Pascal procedure call being built on the call stack of its implementation.
<i>environment</i>	Analogous to a Pascal activation record on the call stack.
<i>unbound instance</i>	An instance where all variables have no value.
<i>backtrack</i>	Return to the state that the program was in when it entered the nearest choice point, and remove the failed clause from the <i>search space</i> for this node; i.e. at this node, the failed clause will not be found in the next search for a matching clause.
<i>unify</i>	Match and make the most general substitution required to make the current goal and the fact or rule head goal identical. A complete <i>unification</i> algorithm is provided in § 2.1.3.

Table 2.1: Definitions of terms used in interpreter pseudo-code.

- If $term_1$ is a variable, then the two terms successfully unify. In this case, however, $term_1$ is instantiated to $term_2$; i.e. $term_2$ is the most general substitution.
- If $term_2$ is a variable, then the two terms unify successfully, with $term_2$ instantiated to $term_1$.
- If both $term_1$ and $term_2$ are compound terms, then these terms unify successfully if both have the same functor/arity and each pair of arguments is also unifiable. Compound terms are unified term-by-term (typically from left-to-right), such that any instantiations made at term i are considered during the unification of term j , where $j > i$; i.e., replacements are made immediately.
- Otherwise unification fails.

For example:

The result after the unification of each term of the two goals

$f(X, g(X), h(u(X)), Y)$ and

$f(A, g(b), h(C), a)$ is

after term 1 - $f(A, g(A), h(u(A)), Y)$ and

$f(A, g(b), h(C), a)$ which is unchanged.

after term 2 - $f(b, g(b), h(u(b)), Y)$ and

$f(b, g(b), h(C), a)$

after term 3 - $f(b, g(b), h(u(b)), Y)$ and

$f(b, g(b), h(u(b)), a)$

after term 4 - $f(b, g(b), h(u(b)), a)$ and

$f(b, g(b), h(u(b)), a)$ which are equivalent.

If term 3 of the second goal was $h(a)$, then the unification of the two terms fails because there is no substitution which makes the structure $h(a)$ equivalent to the structure $h(u(b))$.

2.1.4 Special Features of Prolog

This section will examine three commonly used features of the Prolog programming language: (1) the cut predicate, `!/0`, (2) the or predicate, `or/2`, and (3) dynamic clause assertions and retractions.

The Cut

The cut predicate provides the programmer with the ability to prune the search tree of a given query by inserting the `!/0` goal within the body of a clause. When a `!/0` node is reached, all choice points which have been marked, from the parent node of the cut to the cut inclusive, are discarded, thus reducing search tree. If the program backtracks through a cut, none of the alternative branches of the subgoals to the left of the cut, and none of the parent's alternatives are explored. The nearest choice point must come from a node before the parent of the cut.

The OR

The `or/2` predicate allows the programmer to include a disjunct in the body of a clause. For example, the two clauses:

```
a :- c, d, b.
```

and

```
a :- c, d, e.
```

can be written as:

```
a :- c, d, ( b ; e ).
```

The `or/2` predicate is represented by a semicolon used as an infix operator⁴. In the later case, if `b/0` fails, `c/0` and `d/0` are not needlessly re-evaluated before `e/0` is tried.

The `or/2` predicate can be thought of as a clause which has two alternative paths. In Prolog it is defined as:

⁴A predicate may be written symbolically using an infix, prefix, or postfix operator notation in which case a grouping priority is defined. This is analogous to mathematic operators.

```

or(X,_) :- X.
or(_,Y) :- Y.

```

The underscore is an anonymous or don't-care variable. When an `or/2` node is reached, a choice point is marked. This definition is fine as long as there are no cuts within the `or/2` goal. For example, consider the clause:

```

a :- b, c, ( e, ! ; f ).5

```

From the Prolog definition of `or/2` given above, it is the `or/2` goal which is the the parent of the cut. In most implementations, however, `a/0` would be defined to be the parent.

It should be noted that the `or/2` predicate is not absolutely necessary. If the programmer had wished to cut back to the `or/2` in the previous example, it could have been written unambiguously as:

```

a :- b, c, d.
d :- e, !.
d :- f.

```

If the programmer wanted to cut back to `a/0`, and there are other `a/0` alternatives, then he could have written the following inefficient, but correct code:

```

a :- b, c, e, !.
a :- b, c, f.
a :- ...

```

Dynamic Clauses

Prolog also provides the ability to dynamically *assert* and *retract* clauses; i.e. the programmer can alter the program by adding or removing facts and rules at run-time. For example, if a proof deduces some static information about the program which might be useful later in reducing the proof of another query or in establishing the result of the current query, this “data” can be added as a fact. Dynamic rules

⁵The `','/2` operator has a stronger grouping priority to the `','/2` operator.

are often input by the user to adjust the program to a particular application. In a high level language, like 'C', which is compiled rather than interpreted, similar assertions require run-time compiling and linking; but, for Prolog interpreters, these are relatively simple linked list insertions and deletions⁶.

2.2 The Prolog Machine – Implementation

This section describes a typical von Neumann implementation of an interpreter, therefore, it introduces the fundamental concepts on which compiled Prolog is based. In § 2.1, unification was shown to hold the central role in the deduction of a proof; that is, both the execution and variable bindings are controlled by unification. Efficient matching requires a well thought out representation for Prolog clauses and terms (arguments) which, in turn, influences the overall interpreter implementation. This section begins with a look at the concept of representing the program and continues with how it is stored and how instantiations are made during execution.

2.2.1 Clause Representation

Clauses are stored as tree data structures in the *clause space*. The head and body are each represented by their own graph. The clause predicate defines the root of the head graph, while the body is represented by a series of *,/2* structures which are grouped right to left; i.e. the two rightmost subgoals are grouped together first, then this pair is grouped with third last goal, etc. This is illustrated with an example in Figure 2.4.

The graph descriptions are called *prototypes*; e.g., Figure 2.4 defines the prototypes of the head and body of the clause:

```
green(X) :- gold(X,blue(X,mix([X|a],b)),Y,1,a),
           write(X), write('is green').
```

A clause description is referenced by a composite data structure which includes a pointer to the head prototype and a pointer to the body prototype. The data

⁶ This will be discussed in § 2.2.

```

green(X) :- gold(X,blue(X,mix([X -a],b)),Y,1,a),
            write(X), write('is green').

```

Figure 2.4: Graph representation for a clause.

records which identify clauses of the same procedure are linked together as a list; see Figure 2.5. All *atoms* (constants) in a program are hashed into an *atom table* by the character string which represents them. For example, procedures are hashed according to their predicate symbol (functor/arity). The hash table elements which are referenced by a predicate, include a pointer to the first clause of a procedure (the first element in the list); this is shown in Figure 2.5. The diagram reveals how the predicate of the current goal is found from its corresponding prototype. A predicate node of a graph (e.g. *green/1* of Figure 2.4) is a pointer into the atom table from which the procedure's clause list is found. Every clause in the list after the first, represents an alternative path in the solution. If the first clause fails to prove the current goal, the next clause in the list is tried. When the end of this list is reached, the execution must backtrack.

Figure 2.4 depicts a clause as a graph structure. It indicates that the variables (e.g. *X:(VAR 0)* and *Y:(VAR 1)*) are the only terms which make one "call" to the clause unique from another. Each instance of the same clause has the same general structure, and therefore, when activated, can be represented by a pointer to the clause prototype and a *variable frame*. This is analogous to a Pascal procedure with two local identifiers, *X* and *Y*, which exist only as long as the procedure has not returned. In Pascal the activation record of a procedure invocation (located on the call stack) includes information for returning to the calling procedure, the arguments passed in the call, and the local variables. In Prolog, the local variables are stored in a *variable frame* which is generally implemented in one of two ways: with structure sharing, or structure copying. The next section will explain how primitive terms are represented which leads to a description of how compound terms are defined and managed.

2.2.2 Data Representation

Unification was introduced as the heart of the interpreter in § A.1; it not only provides variable instantiations, but, through its success or failure it actually determines the path of execution. An example program showed Prolog as a typeless language (cref. § 2.1.2), but the unification algorithm assumed that types and values

Figure 2.5: A list of clauses in a procedure referenced through the atom table. The procedure in this example contains two clause alternatives.

are known (cref. § 2.1.3). Without explicit type declarations, the implementation is, therefore, forced to tag the data in order to support the unification algorithm. This section will use the example in Figure 2.4 to discuss data tagging. Two methods of term representation, structure sharing and structure copying, are also presented and compared with one another.

Data Storage

Arguments and terms are represented by compound data objects called *value cells* which consist of two components: a *tag*, and a *value pointer*. The *tag* defines the type of the data, and dictates the contents of the *value pointer*. The extent of information provided by a tag directly relates to the ease with which unification is performed. In general, the following tags are used: (1) atom, (2) number, (3) structure, (4) unbound, and (5) reference. The types of values cells are as defined as follows:

atom — Atoms are program constants. The second component of this type of value cell is a pointer into the atom table. This value pointer references the location returned by the hash insertion function when it is passed the corresponding, printable string of the atom. For example, **cathy** in:

<atom,cathy>

is a pointer to the corresponding atom table element shown in Figure 2.5.

num — A number cell is also quite simple. The value pointers of these cells reference dynamically allocated locations which have been assigned an integer or a floating point number. An example, as it will appear in the text of this thesis is:

<num,4>

structure — A compound term is tagged with a pointer to its prototype. From the example in Figure 2.4, the second term of the **gold/5** subgoal is the structure **blue/2**. A variable bound to this structure would be represented by the following value cell:

Figure 2.6: Value cell for the structure — `blue(X,mix([X|a],b))`.

*<pointer to the blue/2 node of the of the clause body's prototype,
pointer to the clause instance variable frame>*

The clause frame is a vector of value cells which defines the clause instance; i.e. there is a value cell for each variable in the clause. Figure 2.6 illustrates a value cell instantiated to an unbound instance of the `blue/2` term of the example clause. The offsets implied by the variable nodes of the prototype, (e.g. the 0 in `VAR 0`), indicate the corresponding offset into the frame for that variable. This is the structure sharing representation. In structure copying, the value cell and frame will reside together on a *copy stack*, therefore the frame pointer is not required; this is explained later in § 2.2.2.

undef — An undefined cell represents an unbound variable as shown in Figure 2.6. As in Pascal, local variables do not have values until one is assigned to them. When an instance of a clause is created because of a goal call, all variables in

the frame are initially unbound. Unification between the arguments passed in the call and the terms of the head prototype might bind a variable to another type of cell. Only undefined cells can be instantiated to another type and value, therefore any variable bound during an execution path is simply reset to undefined if the program backtracks. Recording the instantiated cells so that they can be later unbound is called *trailing* and will be discussed in more detail in § 2.3.2. In a call to **green/1**, of Figure 2.4, if the argument passed in was an atom cell, (e.g. **<atom,apple>**), the unification of this argument with the first term, (**VAR 0** is the first term of the head prototype shown in Figure 2.4), would change the contents of the corresponding frame value cell to a *reference* cell; this is defined next.

reference — A reference cell does not directly provide a value for the variable it represents. Instead, the value pointer of this type of cell references another value cell. In the example discussed under the *undef* tag, the value cell for **VAR 0** would be:

<ref,pointer to the value cell of the argument passed in the call>

The purpose of a reference cell is to cause all variable instances, bound to the same variable, to *dereference* (eventually reference through a chain of one or more reference cells), the same value cell. Even if the variable passed to **green/1** was itself an unbound value, the **VAR 0** of this clause instance would still become a reference to the argument's cell. When this reference is passed in the call to **gold/5**, any instantiation to this variable would be assigned to the dereferenced cell, therefore, the binding exists only in one location. All instances of the variable dereference to the same cell and, thus, the same instantiation. An example of reference cell linking and dereferencing is given in Figure 2.7.

Unification of two terms begins with a simple tag comparison. If the dereferenced tags match (an undefined tag matches everything), then the values are bound accordingly (cref. § 2.1.3). Otherwise, unification fails immediately and backtracking begins. Term representation is reasonably consistent for the two common ap-

Figure 2.7: Illustration of two variables which *dereference* to `<atom,cathy>`.

proaches to data management called *structure sharing* and *structure copying*. These two methods differ when handling compound terms (structures), and each has advantages and disadvantages with particular applications. Structure sharing is more intuitive with the background given thus far, and is a good starting point for the understanding of, and reasoning behind the structure copying implementation.

Structure Sharing

In structure sharing, the prototype ("source term") of a compound term, is defined by the program; it is a portion of the clause representation. Each clause instance *shares* the same general structure, and differs only in the variable bindings. Like a Pascal procedure invocation, the code is shared, but the local variables are uniquely allocated in an activation record on the call stack. In Prolog the prototype is shared, but each instance has its own variable frame. The value cell of a compound term is therefore, a two pointer data record consisting of a pointer to the prototype, and a pointer to the variable frame.

This approach is analogous to passing by reference in Pascal. The local variables (variable terms) of the "called" clause are maintained in the clause frame which is similar to the activation record of a Pascal procedure invocation (see Figure 2.6). Unification matches and binds the local variables to the arguments passed in the

call. Such bindings change the local cell tags from *undefined* to *reference*, and the cell values become pointers to the dereferenced locations of the arguments.

The example in Figure 2.4 considers a clause with a subgoal **gold/5**. The second term of this goal is the structure:

```
blue(X,mix([X|a],b))
```

which can also be written as

```
blue(X,mix(cons(X,a),b))
```

if the common notation which uses square brackets to represent a list is replaced by the **cons/2** predicate [2, 26]. The prototype of this term is the tree data structure shown in Figure 2.6. A value cell which references this term contains a pointer to the structure and a pointer to a *frame*. The frame, which resides on the *structure stack*, is a vector of value cells *constructed*⁷ to represent the instance of the clause variables. For this example, **X** and **Y** are the only variables appearing in the clause, and therefore the frame consists of two value cells which were renamed **VAR 0** and **VAR 1** respectively in Figure 2.6; i.e., variables are indexed by the location of their first occurrence. The **blue/2** structure contains only the **X:VAR 0** variable.

If the invocation of the goal **gold/5** is viewed as a call from **green/1**, the five arguments passed in the call are those given in Table 2.2.

In § 2.1.3 the act of substituting across a structure as a result of unification was discussed. Here, one can see that physical substitutions are not really performed. The **X** variable is simply represented by the same value cell across the entire clause, therefore, substitutions are automatic. For example, **write/1**, which is a subgoal called later in the example clause, is passed the same value cell for **X** that was passed to **gold/5**. The variable frame cell for the local value of **X** also dereferences to this very same cell. When any occurrence of **X** is instantiated, only one value cell (the last in the dereferenced chain) is updated. Because all occurrences of this variable dereference to the updated cell, all occurrences are instantiated at the same time. Reference cells, therefore, simplify the means by which the program state is restored

⁷This is the word used to refer to the run-time creation of a term instance.

(1)	<i><X was passed into green/1 therefore the same value cell is passed to gold/5></i>
(2)	<i><pointer to the prototype in Figure 2.6, pointer to the frame for the green/1 clause ></i> The contents of VAR 0 in the variable frame is: <i><ref, pointer to the value cell of X></i>
(3)	<i><undef, undef></i>
(4)	<i><num, 1></i>
(5)	<i><atom, a></i>

Table 2.2: Arguments passed to **gold/5** by **green/1** in a structure sharing implementation.

on backtracking because only one cell has to be reset if the variable was instantiated during the execution.

Another example, taken from [13] defines a prototype for **t/2** as shown in Figure 2.8. This term appears in the source as **t(p(X),q(p(X),Y))**, and the frame is a vector of two value cells; one for VAR 0 and one for VAR 1. If two terms were constructed from the clause containing this structure, they would both point to the same prototype; they would “share” the prototype, but each term would have a frame pointer corresponding to its instance of the clause. An example program which further illustrates structure sharing and the construction of frames on the structure stack is considered in § A.2.

Structure Copying

It is possible for a term to outlive the clause instance from which it was created. Structure copying takes advantage of these situations to achieve gains in memory efficiency. As one might expect, the call stack (execution stack) of a recursive depth first search strategy can become quite large. Structure copying takes an approach which allows a procedure to return immediately to its caller without destroying

Figure 2.8: Prototype for the structure — $t(p(X), q(p(X), Y))$.

the variables it has created. The method is much like one used in Pascal where a procedure which is to create a long lived object does so in the heap instead of in local variables, (on the call stack), and then returns a pointer to this object.

This will occur, for instance, when the last alternative of a procedure is called and only one solution is required. In this case, the variables represent the only information which is required for the proof to continue. If the clause produces only one solution then no choice points are defined, and therefore the control information, which basically describes the direction of the proof when backtracking, is not needed. The control information would include, for example, a pointer to the parent goal and a pointer to the next alternative clause. The usefulness of the variables have outlived the remaining contents of the clause instance.

In structure sharing a clause instance requires a prototype and a variable frame. In structure copying the clause is still represented by its prototype and a variable frame, however, the frame locations hold only single pointers. Instead of pointing to the prototype when a compound term is to be constructed, a term *copy* is made; that is, a “real instance” (the copy) is built from the “virtual instance” (the prototype) on

the *copy stack*. Any variable which might outlive its clause instance is constructed on the copy stack and referenced (with a reference cell) by a local variable. If passed as an argument, a reference cell which points to the copy stack is passed. These variables are not defined by value cells of the clause variable frame from the point of view of any other goal instance, but rather through the copy stack. The copy stack must stay in tact during the forward execution of a proof, while the call stack space of a *determinate* clause (one with no alternatives and thus no control information) can be reclaimed.

When a frame value cell is constructed to represent a compound term, a reference cell is made which points to the location at the top of the copy stack. The copy stack is a LIFO stack of value cells where prototype copies are made whenever a compound term must be constructed. Therefore, the variable frame entry for a compound term references the copy stack where a value cell tagged with the structure's functor/arity is found. This tag, like the node of a prototype, references an atom table entry. The value cell which indicates the type of the structure is followed immediately by the value cells defining each argument of the structure.

Using the same example of the previous section, the arguments passed to `gold/5` by `green/1` in a structure copying implementation are shown in Table 2.3.

If another goal is matched with this clause, it will have its own instance of the structure built on the copy stack. Value cells for a structure copying implementation are generally considered to be simpler because there is only one pointer required. An illustration of structure copying is provided in § A.2.2, and a complete comparison between structure sharing and structure copying is drawn in § A.3.

2.3 An Interpreter Architecture

The execution of a query requires several stacks (the control, the copy, and the trail stack), a set of value cells in which to pass the arguments of a goal call, the clause prototypes which are stored in the clause space, an atom table, and several global pointers. On top of all these global data structures, each environment on the control stack has a variable frame, and its own set of pointers which represent

(1)	<i><same argument passed to green/1; X></i>
(2)	<i><ref, pointer to the initial top of the copy stack at i in the diagram below></i> the copy stack is: i+8: <i><atom, a></i> i+7: <i><ref, pointer to copy stack location i+1></i> i+6: <i><struc, cons/2></i> i+5: <i><atom, b></i> i+4: <i><ref, pointer to copy stack location i+6></i> i+3: <i><struc, mix/2></i> i+2: <i><ref, pointer to copy stack location i+3></i> i+1: <i><ref, pointer to the value cell of X></i> i: <i><struc, blue/2></i> The nested structures, like the cons/2 in mix/2 , and the mix/2 in blue/2 have been constructed after their parents in this example.
(3)	<i><ref, pointer to the new top of the copy stack at i+9></i> which contains: i+9: <i><undef, undef></i>
(4)	<i><num, 1></i>
(5)	<i><atom, a></i>

Table 2.3: Arguments passed to **gold/5** by **green/1** in a structure copying implementation.

the execution state on invocation of the goal instance. Figure 2.9 illustrates these objects and the correspondence between them⁸. During a proof the stacks grow as a result of unification, and shrink, for the most part, when backtracking. Prototypes, stacks, the atom table, and arguments are all accessed through pointers. The stack pointers define the top of the stacks, and are used during forward execution. The choice points of the execution are used when backtracking. This section will discuss these objects, their relationships, and their contents during forward and backward execution of a proof assuming a structure copying approach to data management.

2.3.1 Arguments

The arguments passed in each goal call are assigned to specific value cells designated for the arguments of the current procedure call. As shown in Figure 2.9, there is only one set of argument cells, therefore, they are destructively assigned the appropriate contents for the current call.

2.3.2 The Stacks

There are three stacks in an interpreter implementation: the control stack, the copy stack, and the trail stack as shown in Figure 2.9. Each carries information which is very important to a goal's proof. They are defined and used as follows.

The Copy Stack

The copy stack was covered in detail in section 2.2.2 and Appendix A.2. It is the LIFO stack on which all compound term instances are built, and on which variables which might outlive their clause instance are *globalized*. Two global pointers are kept which allow the copy stack to be used and maintained. They are:

Top of Copy Stack — The top of the copy stack pointer indicates where the next global variable is to be created.

⁸The diagram shows a potential setup for the recursive call of `member/2` which is defined in Figure A.5.

Figure 2.9: Data objects and pointers in an interpreter implementation.

Choice Point of Copy Stack — This pointer references the copy stack where the last environment with an alternative clause (the nearest choice point) was invoked. When the execution backtracks, the pointer to the top of the copy stack is reset to the value stored in this pointer. All terms built since the nearest choice point, (built during the search down a failing path), are removed with this simple pointer assignment. Variables which are located below this point on the stack, and are instantiated during execution must be trailed. To reset the execution completely, instantiations to copy stack value cells which are not removed by resetting the top of copy stack pointer, must be unbound. In order to find these cells their locations are saved on the trail stack.

The Trail Stack

The trail stack is used to record variable instantiations. It is a LIFO stack on which pointers to the value cells of instantiated variables are stored. As mentioned above, only cells which were created before the nearest choice point need to be remembered. During unification, if an instantiation is made to a variable on the copy stack, the location of its cell is compared to the “choice point of the copy stack” pointer. If the cell was created before the nearest choice point’s instance was created, then this cell is trailed. To trail a variable, a pointer to its value cell is stored on the trail stack as shown in Figure 2.9. When backtracking to the nearest choice point, any bindings trailed since this instance was invoked, are undone; i.e., the value cells are reset to `<undef, undef>`. The global pointer associated with the trail stack is:

Top of Trail Stack — The top of the trail stack represents the location of the next trailing operation, and the point to which trailing has extended (how far to read when unbinding instantiations during backtracking).

The Control Stack

The control stack is the clause environment stack. Whenever a procedure is called to solve a goal, an instance of the first clause in the procedure is created on the control stack. These environments contain the variable frame corresponding to, and

indexed by the clause prototypes, the argument registers at the time the environment was created, and a set of pointers which allow the execution state to be reset on backtracking or to continue through a clause's subgoals as described below:

Arguments — The arguments passed to this clause are saved. When backtracking to an environment, the global argument registers, which are destructively assigned new values on each call, must be reset before invoking the next alternative clause in this procedure.

Top of Copy Stack -- The top of the copy stack on entry into an environment is saved. This value is assigned to a global pointer of the same nature when backtracking which automatically removes the cells constructed on the copy stack by the failing path of goal calls. If and when this environment becomes the nearest choice point, the value stored here is assigned to the global pointer which indicates the top of the copy stack.

Top of Trail Stack --- The top of the trail stack on entry into an environment is also saved so that variables trailed after this point can be unbound on backtracking. This value represents the starting point for the clean-up process of the copy stack. All bindings trailed between this point and the current top of the trail stack are unbound.

Continuation Pointer — This is a pointer into the clause body prototype. It points to the node representing the next subgoal which must be proven. For example, when the goal **gold/5** of the example in Figure 2.1, successfully returns to the **green/1** environment, the continuation pointer is indicating that **write/1** is to be called next. Before making the call, this pointer is set to reference the node of the second **write/1** call. This pointer is used to determine where the proof is to continue when the current subgoal completes successfully. In effect, the resolution stack of § A.1 is segmented throughout the control stack with each parent keeping track of where, in its conjunction of subgoals, the proof has reached.

Parent Environment — A pointer into the control stack to a goal's parent environment is saved to allow a successful proof to return correctly. The pointer is also used control backtracking through a cut. The nearest choice point of the parent of a cut becomes the overall nearest choice point when backtracking goes through the cut predicate.

Environments Nearest Choice Point - This pointer is used when backtracking to reset the global choice point.

Alternative Pointer — This is a pointer into the clause space. It points to the procedure list element representing the next alternative, (see Figure 2.5).

One global pointer is used in the maintenance of the control stack:

Choice Point — The choice point pointer references the control stack at the environment where the nearest choice point resides. The nearest choice point of this environment, therefore, becomes the new global choice point when backtracking returns to this environment.

2.3.3 The Clause Space

The clause space is an area where linked lists of clause prototypes (linked according to the functor/arity) are stored. This was illustrated in Figure 2.5, and is also shown in Figure 2.9.

2.3.4 The Atom Table

The atom table is a hash table of program constants. The table provides printable strings for the constants which are being represented, and pointers into the clause space for a given functor/arity.

2.3.5 Effects of Execution

An interpreter architecture has been defined as a set of data structures and pointers. The effects of a query proof on the state of this design are explained in § A.4. It

will become clear from this example that Prolog interpretation uses a great deal of memory to build and keep all of its stacks during execution. One technique used to reduce the memory requirements of a proof is called *tail recursion optimization*. A description of this feature is given in § A.5.

2.4 Warren's Abstract Machine - WAM

D.H.D. Warren introduced an abstract machine for Prolog (WAM) consisting of a set of intermediate language instructions and the data objects on which they operate [30, 16, 22, 4]. The clause prototypes of the interpreter design are replaced by specific instructions which essentially correspond to the nodes in the graph of a prototype. The instructions represent an alternative description of the source code, and are produced through the compilation of a program. The data objects in WAM are fundamentally the same as those of an interpreter implementation: WAM uses the same stacks and similar information pointers (cref. § 2.3). One significant change, however, is that procedure pointers supplied by the *atcm* table, are now pointers into the code space instead of the beginning of a linked list of prototype pointers.

The WAM approach attempts to improve interpretation by increasing the speed of execution and decreasing the memory requirements. One of the reasons compilation improves the performance is because specific information can be extracted about each goal and term. With this knowledge, instructions are created which only use the relevant sections of the general unification algorithm. If less steps are taken to unify two goals, then the efficiency of the execution has been enhanced. This information can be exploited in other, less intuitive ways which will be discussed in this section.

2.4.1 Data Representation and Classification

It was shown in § 2.2.2, that structure copying was generally more memory efficient than structure sharing. WAM is a structure copying implementation where each value cell uses only one word of memory consisting of a two bit tag and a value pointer. An "undefined" cell is represented by a reference cell whose value pointer

points to itself, but all other tag and value pointer combinations are consistent with those of the interpreter model (cref. § 2.2.2).

The variables of a clause are classified during compilation. The classification indicates the lifetime of a variable which, in turn, allows memory to be allocated in the most efficient way as follows:

temporary — A variable is classified as temporary if its first occurrence is in either: (1) the head of the clause, (2) a structure, or (3) the last subgoal, and the variable does not occur in more than one goal of the clause unless it is shared by the head and first subgoal [30]. Therefore, all variables of facts, and rules with only one subgoal are temporary variables. Temporary variables are stored in the argument registers; i.e. the local instance of a temporary variable is placed in an unused argument register. There are a fixed number of argument value cells (registers) in this implementation. This is consistent with the interpreter model given earlier. Those cells which are not required for a call represent wasted space. WAM uses them to store the local instances of variables which need only exist as long as the current call is being proven (recall that all variable instances are stored in the control stack variable frame in the interpreter design). In effect, temporary variables are being referenced on only one subgoal branch of the parent node, therefore, they need to exist only during the proof of that particular branch. These variables do not need to exist in the clause environment after the subgoal returns. Structure variables, which might actually become referenced later in the proof, are already “permanently” stored on the copy stack, therefore only the reference cell to the copy stack is really temporary.

permanent — All other variables which are not temporary, are permanent. These variables exist in their parent’s control stack environment, therefore their lifetime is the same as that of the clause. A permanent variable is one which is used in the proof of more than one subgoal of a clause, therefore it must exist during the call to all goals to which it is passed as an argument. These value cells are placed at the top of the clause environment in such a way that *environment trimming* is possible. This means that the variable with a

```

qsort([],R,R).
qsort([X|L],R0,R) :-
    split(L,X,L1,L2),
    qsort(L1,R0,[X|R1]),
    qsort(L2,R1,R).

```

Figure 2.10: `qsort/3` predicate.

shortest life time is placed at the top of the stack, and discarded as soon as possible.

unsafe --- The unsafe classification is a special case of the permanent variable. This title is given to a permanent variable which did not first occur in a structure or the last subgoal of the clause. This type of variable might need to be *globalized* by binding it to an undefined value cell of the copy stack. An unsafe variable might outlive its clause instance, and globalizing avoids having dangling references when the unsafe variable's parent environment is reclaimed on a determinate exit. Unsafe variables are not dealt with until their last occurrence is required for a call. At that time, if the variable has not been bound to a senior variable, it is globalized.

remaining variables — These are variables which occur in only one subgoal which is neither the first nor the last, and did not first occur in a structure. They do not require any frame cell because they can be handled simply from the argument register to which the first occurrence is assigned.

For example, the `qsort/3` predicate shown in Figure 2.10 would have its variables classified as follows:

- The first clause is a fact, therefore, all variables occur in only one goal, hence `R` is a temporary variable.
- The variables `X` and `R` of the second clause appear in two subgoals, and both first occur in the head goal, therefore they are permanent variables.

- The head and first subgoal together, count as one subgoal. **R0** and **R1**, appear in two subgoals, therefore they are also permanent variables.
- **L1** and **L2** reside in more than one subgoal, but the first occurrence of both is neither within the head goal or a structure. These variables are unsafe.

The structure copying implementation, along with the use of compile-time variable classification provides a memory efficient approach to data representation and storage.

2.4.2 Instruction Set

There are five groups of instructions in the WAM instruction set: *match*, *argument*, *unify*, *clause indexing*, and *control* instructions. For Prolog execution, these correspond to the four procedural steps shown below:

(1)	Search for an appropriate clause to prove the goal.	<i>index</i>
(2)	Match the head goal with the incoming arguments.	<i>match and unify</i>
(3)	Define the arguments of the subgoal calls.	<i>argument and unify</i>
(4)	Transfer control to the subgoals and act on their result.	<i>control</i>

One of the most recognizable benefits of compiling Prolog is the simplification of unification which is exploited by the *match*, *argument*, and *unify* instructions. The compiler determines the structure of a clause at compile time. Each node of the interpreted prototype becomes a specific specialized instruction which performs only the parts of the unification algorithm which are applicable to that node. The WAM instructions represent a very efficient way to unify a clause. For instance, using Figure 2.4, the prototype for the head of the clause, **green(X)** consists of

two nodes. In WAM these nodes are represented by two instructions: (1) an index instruction which indicates what to do if the proof fails with this clause; in this example there are no alternatives, therefore the goal fails if this clause causes a failure, and (2) a match instruction to unify the first term of the clause with the first argument passed in the call; i.e:

(1) **try_me_else fail**

(2) **get_variable(local variable location, argument register number)**

The local variable location, depends on the classification of the variable. The example clause is:

```
green(X) :- gold(X,blue(X,mix([X|a],b)),Y,1,a),
           write(X), write('is green').
```

The variable **X** occurs in two subgoals, therefore it is a permanent variable. Its local variable location is, hence, in the variable frame of the environment. The body of the clause would again compile into approximately one instruction per node of the corresponding prototype. These instructions are given below as an example, however, each group of instructions will be explained in more detail in the sections which follow.

(3) **put_value(local variable location, argument register number)**

— an argument instruction which places **X** in the first argument register in preparation for the call to **gold/5**.

(4) A series of argument and unify instructions which build the **blue/2** structure on the copy stack, and place a reference to it in the second argument register.

(5) **put_variable(local variable location, variable classification, argument register number)**

— an argument instruction which places **Y**, an unbound temporary variable, in the third argument register. **Y**'s local variable would be in argument register number six; note that this is the first used argument cell in this call.

- (6) `put_const( 1, argument register number)`
 — an argument instruction which places a value cell representing the number 1 in the fourth argument register.
- (7) `put_const( a, argument register number)`
 — an argument instruction which places a value cell representing the atom `a` in the fifth argument register.
- (8) `call gold/5`
 — a control instruction which calls `gold/5`.
- (9) — an argument instruction which places `X` in the first argument register in preparation for the call to `write/1` as in (3).
- (10) `call write/1`
 — a control instruction which calls `write/1`.
- (11) `put_const( 'is green', argument register number)`
 — an argument instruction which places a value cell representing the atom `'is green'` in the first argument register.
- (12) `execute write/1`
 — a control instruction sets the global continuation pointer (a pointer into the code space) to the next instruction of the parent of `green/1`, and calls `write/1`.

As shown in the example above, the compiler *types* each argument of a clause; for example, as a variable, a constant, or a structure. Variables are then classified as discussed in § 2.4.1. This information allows several simplifications to the general unification algorithm to be made. Each match, argument, and unify instruction maps into specific unification steps. The simplifications which can be made are:

1. The general unification algorithm can be written as a set of specialized functions where each considers a particular type of argument. For example if a clause argument is a constant, then the dereferenced incoming argument must be the same constant or an unbound variable. The WAM compiler will replace a constant term in the head of the clause with the instruction :

`get_const(actual constant, argument register),`

while a constant term in the body of a clause can be managed as a specific assignment to an argument register by replacing it with the instruction:

put_const(*actual constant, argument register*) .

Compile-time typing can be extended so that, for example, constants are further split into integers, floating point numbers, and atoms.

2. Matching the first occurrence of a variable in the head of a clause implies that an unbound variable is to be unified with the incoming argument, therefore a specific instruction which simply binds the local variable to the senior incoming argument can be used. This instruction is:

get_variable(*local variable, argument register*) .

3. If the first and only occurrence of a variable is in the head goal, then there is no need to perform any unification operation, therefore, no action is taken in the WAM implementation.
4. For the first occurrence of a variable in a body goal the only requirement is to create an undefined variable which is referenced by the argument register.
5. Arguments of a structure can be classified specifically and treated in a similar manner to all other clause terms. Because all structures reside on the copy stack, these arguments are replaced by *unify* instructions which act on a referenced copy stack location.

Descriptions of the five groups of instructions follows with emphasis being given according to their impact on the PIC implementation.

Get, Put and Unify Instructions

The instructions used for unification in the WAM implementation are listed and defined in § A.6. A description of the unification of **blue**(X,mix[X|a],b), which was avoided in the previous section (cref. step 4 of the example above), is included.

Control Instructions

Control instructions implement the global pointer updates and the environment creations and deletions (cref. § 2.3). This part of the WAM implementation is not used in the PIC design, but is explained in § A.7 because it will reveal how the PIC implementation is restricted by the 'C' control structure.

Indexing Instructions

The last set of instructions are the clause indexing instructions, which have two major functions: (1) to optimize the selection of clause alternatives (skip over clauses which will never unify with the incoming arguments by comparing the type of the first argument with that of the first term in each clause), and (2) to specify control in the selection of clause alternatives.

Function (1) The purpose of the *switch* instructions is to avoid unnecessary unifications by defining the expected type of the first argument. For example, if the compiler recognizes that the first term of the clause is a constant, and the the first incoming argument dereferences to a structure, then the compiler has predetermined that unifying these goals will fail, and that the code can be skipped. This sort of classification indicates where determinate procedure exits are present. For example, if the first clause in a procedure matches its first argument with the empty list atom, `nil`, and all of the other clauses match with different views of the list structure, then, if the first clause is invoked because the incoming argument was an atom, the procedure will succeed or fail determinately. These actions are encoded with `switch_on` instructions which essentially represent a jump table to different points in the code according to the type of the first incoming argument. An example is given in § A.9.

Function (2) The other function of the indexing instructions is to control the selection of clause alternatives. This is encoded using `try`, `retry`, and `trust` instructions which represent the first, an intermediate, or the last alternative respectively. These instructions might appear at the jump locations of a

`switch.on` case and indicate the appropriate alternatives. The specifics are given in § A.8.

2.4.3 Instruction Optimizations

Warren indicates that some instructions represent null operations. For example, moving an incoming argument to the same register for the next call is not necessary. He states that his compiler takes “pains” to maximize the number of such instructions to save memory as well as time. Value cells on the control stack are arranged so that they can be discarded as soon as possible. Therefore a permanent variable which is used only in the first subgoal call is placed at the top of the stack and its location is reclaimed before the next subgoal call. Several occurrences of null instructions are found in the example used in § A.9.

2.5 Concluding Remarks

The PIC implementation combines these or variations of these concepts in a manner which produces a strictly procedural *Prolog in 'C'*. The next chapter reveals the basic format of the translator output, and the details of unification in PIC follow in Chapter 4.

Chapter 3

General Design of Prolog in 'C'

The introduction stated the following thesis objectives:

1. to provide a dual logic and imperative programming environment which generates a single, common language, target;
2. to define PIC, *Prolog in 'C'*, in a simple and consistent manner, therefore, providing the opportunity for the user to tune the translator output, or hand-compile his own functions to interface with the translated code;
3. to provide a standard calling sequence for PIC procedures in order to simplify interfacing the different languages.

The first objective demands the selection of two input languages, one imperative and one declarative, and a target language. As inputs, Prolog and 'C' have been chosen because of their accessibility. The output language chosen is again 'C'. This decision is based on the availability of the language, and its flexibility with respect to pointer management and manipulation. However, the focus of this objective is to provide a dual language programming environment. This will be satisfied by a system with the configuration shown in Figure 3.1 which includes three general areas: (1) inputs to the target software generators, (2) the generators, and (3) the target software. Each of these sections is subdivided as follows:

The Generator Inputs — The bottom tier of Figure 3.1 defines the inputs to the generators. A user supplied Prolog program is input to the Prolog 'C'

Target Program Components

Generator Inputs

Figure 3.1: Prolog and 'C' Multi-language Development Environment.

translator (PICG) shown in the middle level, and is output from there as PIC code. This code can be integrated into a target program, where it is called through a 'C'-Prolog interface (CPI) function. The interfacing function is constructed from the information provided in two files, CPIDEF and PICDEF.

The Software Generators — The middle level of blocks in Figure 3.1 represent the generators which produce the CPI functions and the PIC code shown in the top level. The arrows at this stage of the diagram trace the flow of the inputs and outputs to these generators. The PICG, (PIC generator), takes a user written Prolog program as input and produces an application specific definition file, PICDEF, and the corresponding PIC code as output. The CPIG, (the CPI function generator), requires as inputs a user supplied definition file, CPIDEF, and the PICDEF file to produce the CPI functions.

The Target Software — The target software is represented by the three blocks at the top of Figure 3.1. The left-most block contains the 'C' code supplied by the programmer. The center block illustrates the interface functions which are used to invoke the PIC functions depicted by the rightmost block. The arrows drawn at this level indicate the flow of control during execution of the target program.

This chapter will begin by expanding these statements into a set of minimal requirements which establish the basis of the PIC representation. The fundamental control strategy is designed, and this is followed by a description of the remaining high level features found in *Prolog in 'C'*.

3.1 Requirements

Requirements for the 'C' code

Figure 3.1 shows all calls in the target software starting in the user supplied 'C' program, therefore, the target program mainline code must be part of the original 'C' program. The target software will always begin in a 'C' environment from which a CPI function can be called to execute a PIC function. Passing through

a CPI function changes the program state from its current environment to a new *Prolog environment* (one which can access a corresponding set of Prolog data structures). To meet the objective of understandability, the main requirement is simplicity, therefore, the initial, top level 'C' environment will not be able to access any Prolog variables. All communications starting from a 'C' environment will be managed by the CPI function. This is illustrated in Figure 3.2.

Requirements for the CPI functions

The 'C' -Prolog interface is restricted in the sense that 'C' code cannot access Prolog variables, thus within a Prolog environment, data can be represented in a form convenient to the execution of PIC. The purpose of a CPI function is to provide a means of communication between the languages which basically involves: (1) the creation of the Prolog environment, and (2) the transformation of 'C' arguments into Prolog terms as a result of a 'C'-Prolog call, and vice versa on return. The transformations are defined by the contents of the files CPIDEF and PICDEF.

Requirements for the PIC procedures

To produce easily integrable PIC, Prolog procedures must translate into individual 'C' functions which are self-contained and simple to invoke. The third objective stated that all PIC functions are to be invoked with a consistent calling sequence, therefore, the arguments passed in a call are to be stored in an incomplete structure like a linked list, or restricted to a given number. The later was chosen for the preliminary implementation. Simplicity and consistency are requirements of the second objective which are supported by using the WAM approach to term representation and compilation. For example, the unification simplifications exploited through term classification will cause WAM-like get, put and unify instructions to be generated. These will represent 'C' function calls which will operate on a structure copying term representation. The complete unification procedure, which includes dereferencing, binding, and trailing, is thus hidden behind these instructions, and the user need only understand the classification procedure and instruction selection of WAM to write hand-compiled PIC. Finally, PIC must be reasonably time and

Figure 3.2: Mode switching allowed in the target program.

space efficient. This is partially satisfied by using Warren's optimizations on unification, but can also be improved by considering determinate exits and tail recursive clauses.

Requirements for the CPIG

The CPI generator will apply: (1) type definitions and PIC function declarations supplied by the 'C' programmer in the CPIDEF file, and (2) the Prolog specific information extracted by the PICG and written to a PICDEF file, to the creation of the functions which will interface the 'C'—Prolog calls. This thesis deals mainly with the PICG implementation, however, some suggestions for the CPIG functionality are given in Chapter 4. The CPIG is a separate, incomplete research project.

Requirements for the PICG

The PICG must be flexible enough to minimally handle the following important Prolog language features as inputs: pure logic, negative logic, the cut and or operators, and dynamic fact assertions and retractions. The translator must also be able to output suitable PIC files.

Requirements for the input Prolog

For the sake of simplicity, there will not be a background interpreter in the target system for executing Prolog code which is difficult to translate into PIC code (e.g. dynamic clauses). The PIC generator input must conform to the limitations of the translator.

Requirements for the CPIDEF and PICDEF files

The CPIDEF and PICDEF files must include all of the information required to produce the interfacing functions. The specific requirements for the contents of these files are defined by the design of the CPIG. Chapter 4 will provide a description of the necessary information.

Requirements shared between the PIC and CPI functions.

Because 'C' is the target language, simplicity and understanding are enhanced by producing typical 'C' functions from the Prolog procedure definitions. As opposed to emulating the interpreter in 'C', PIC functions are invoked through the normal call and return statements of standard imperative 'C' programming. The 'C' call stack is used to house the predicate environments (a separate control stack is not being introduced when a Prolog environment is created). Since all functions called within a Prolog environment follow a standard calling sequence, Prolog 'C' calls are easily integrated into this implementation.

This approach will cause the Prolog environment to disappear when the execution returns through a CPI function, and will allow choice points to be created only immediately before their execution (a choice point environment can not be created at the top of the control stack because the currently executing function must be running at the top of this stack). All other PIC implementation structures, (e.g. the copy and trail stacks), will not exist if a Prolog environment is not currently running. The CPI function will dynamically create what is necessary to obtain a unique self-contained Prolog environment.

Figure 3.2 illustrates the extent to which the programming languages are to be integrated. Each box below a CPI call represents a separate Prolog environment whose variables are only available to functions within the same environment. A 'C' function in a Prolog environment represents a goal in the Prolog code which is to be hand written in 'C' with the appropriate PIC calling sequence. Hand-compiled PIC does not have to access Prolog variables and/or backtrack, in which case the function is strictly 'C' code being called from the Prolog code, and is therefore, represented as a state change in Figure 3.2. Note that this new 'C' environment is contained completely within the Prolog environment where it was invoked, and that these functions can call any PIC function, or create a new Prolog environment through a CPI function call. If a new Prolog environment is established, the calling environment must wait for the invoked code to complete and return. Returning from a Prolog environment implies that all of the necessary implementation structures have been deallocated, and that the environment and all of its information, (except

for that passed back through return parameters), have been lost.

The requirements which have been listed here define the basis for, and guide the design of, procedural *Prolog in 'C'*.

3.2 The Basic PIC Control Strategy

One of the requirements for the PIC code stipulates that each Prolog procedure is to be a separate, consistently styled function whose representation must include, and be able to process all clause alternatives. The chosen design is given in Figure 3.3. Each clause is a case in a switch construct indexed by the `backtrack_level` which is zero for the first alternative. With this defined, the next step is to develop the basic control strategy for execution. This section establishes a definition of *Prolog in 'C'* which satisfies the abstract model of Prolog given in § 2.1 using the standard features of the 'C' programming language. The result will employ an approach called *continuation passing* [17]. The analysis taken to reach this decision, and an explanation of this technique constitute the remainder of this section.

A first approach to representing the flow of control of *Prolog in 'C'*, uses standard imperative techniques. This is similar to WAM in that each clause is expanded into a set of sequential procedure calls. For example, given the simple program of Figure 3.4, procedure **a** would compile into the following series of call instructions:

```
call b;
call c;
call d;
```

The flow of calls for this example is shown in depth-first, left-right ordering in Figure 3.5 where the arrows illustrate the direction of the data flow; i.e., the returned status. When **d** fails, the execution must return to the last choice point, which is in procedure **c**, and try the next alternative. The PIC procedures, **a**, **b**, **c** and **d**, are required to be encoded as 'C' functions with the general format given in Figure 3.3. When **c** was first called, the code of the first clause was tried (**CASE 0** of the **SWITCH** block was executed). From this clause procedure **e** was called, and it returned a successful status to **c**, which was, in turn, returned to **a**. The program continues

```

success = FALSE;

WHILE (success == FALSE) {
    SWITCH (backtrack_level) {
        CASE 0:  Try clause 1;
                IF successful
                    success = TRUE;
                ELSE {
                    unbind variables;
                    increment backtrack level;
                }
                break;
        CASE 1:  Try clause 2;
                .
                .
                .
        CASE n:  Clean up execution state;
                RETURN failed;
    }
}

RETURN success;

```

Figure 3.3: Pseudo-code for the basic PIC procedure definition.

```

a :- b,c,d.
b.
c :- e.
c :- f.
d :- fail.
e.
f.

```

Figure 3.4: Example program for illustrating control flow.

with **a** calling **d** which returns a failure status. The next instruction in procedure **a** is now responsible for managing this failure. However, the information required to invoke the next alternative of **c** is found in the **c** procedure not the **a** procedure. This approach cannot be taken because choice point information cannot be reached.

In the WAM implementation (cref. § 2.4) this worked because a choice point environment would have been created by a **try_me_else** instruction before the first alternative of procedure **c** was tried. Such an environment contains all the information required to execute its corresponding clause. In the PIC implementation, the requirement to use the 'C' call stack for procedure invocation takes away the opportunity to dynamically establish an alternative environment at the top of the control stack. The PIC control stack is being managed strictly by the 'C' engine which builds call environments (activation records) at the top of the stack, and removes the top-most environment when a **RETURN** instruction is reached. All clauses of a procedure, (all choice point environments), use the same call stack space allocated in the initial procedure call.

In the example, if procedure **d** had returned its failing status to procedure **c**, then the **WHILE** loop of Figure 3.3, along with the advancement of **backtrack_level**, would have moved the execution on to the next alternative of **c**. In order to revisit choice point functions when backtracking, a *resolution stack* of next goal calls is required to direct the flow of control. Like the interpreter model, goals are called in

Figure 3.5: Flow of procedure calls using a typical imperative approach.

the order in which they appear on the resolution stack, and although the parent is not invoking the goal, it does define the position of it's subgoals on this stack.

Fortunately 'C' provides a feature which allows functions to be called through a pointer, therefore a function pointer and the parameters to be passed in the call can be stored and later used to invoke that procedure [12]. These pointer and parameter pairs, referred to as *closures* in [17], are stored in a linked list structure called a *continuation* which is essentially the resolution stack of Figure 2.3. Figure 3.6 illustrates the implementation of a continuation list of function closures which is used as follows:

Facts— Once a fact is successfully unified with the arguments passed in the call, the next goal to be invoked is the first goal of the continuation list. The tail of the list becomes the updated continuation.¹ This is analogous to popping the top goal of resolution stack.

Rules— Each rule calls the procedure of the first subgoal in its body definition, and passes the continuation list which has been prepended with the remaining

¹Tail is a term used to describe a list consisting of all but the first element of the original list

local contin: a:-b,c,d.

Figure 3.6: Illustration of a continuation list of function closures.

subgoals of the clause. This is analogous to pushing the subgoals onto the top of the resolution stack (cref. § A.1).

A closure is formally defined as a procedure which has some or all of its arguments locked prior to being executed [15], or alternatively, as a function and an environment of bindings [25]. A continuation is also a general term used to describe a data structure which defines the future course of a computation [29]. This approach to procedure invocation described in this thesis is managed by *continuation passing* which has been used in studies on language semantics and in other versions of compiled Prolog [17, 32]. The term refers to the concept of passing a continuation argument in a procedure call [1].

If, in the above example, procedure **e** had called **d**, then the failure status would have reached **c**, and there would have been no problem determining the next alternative: **e** would relay the failure to its caller, **c**, because it has no alternatives of

its own. The return of a failure status to **c** which is executing in **CASE 0**, causes the **backtrack_level** variable to be incremented, and any bindings made since the creation of **c**'s environment to be removed before the execution loops to the second clause of the procedure. With this design, procedure **a** would include the following set of instructions, where **continuation_list** is a variable passed in the call to procedure **a**, and **prepend** is a function which is analogous to the Lisp function **cons**²:

```
continuation_list = prepend(d's closure, continuation_list);
continuation_list = prepend(c's closure, continuation_list);
status = b( arguments, passed structures, continuation_list);
```

The subgoal calls are added to the continuation list right-most first, so that the first closure of the list represents the first call to be made, as did the top element of the resolution stack for the interpreter implementation given in Figure 2.3. Using the same example program, the execution of the query **a** would flow as shown in Figure 3.7, and described by these steps:

1. Procedure **a** calls **b** passing it a continuation list equal to the continuation list passed into this procedure prepended with closures for calling **c** and **d**.
2. Procedure **b** is a fact which succeeds, therefore, it must make the call represented by the first closure of the continuation which is a call to procedure **c**. The tail of the continuation is passed in the call.
3. Procedure **c** calls **e** (the first subgoal of its first alternative) and passes on its received continuation.
4. Because procedure **e** is a fact, it makes the call defined by the first element of the continuation list which is a call to **d**.
5. Procedure **d** fails, therefore it returns a failure status to its caller, **e**.
6. Procedure **e** is not a choice point, therefore the fail is relayed to **e**'s caller, **c**.

²This function creates a new list element with the value of the first argument, appends to it the list given in the second argument, and returns this new list

Figure 3.7: Flow of procedure calls using continuation passing.

7. Procedure **c** is a choice point, therefore, the second case of its **SWITCH** block is reached; refer to Figure 3.3. Here, **f** is called and passed **c**'s originally received continuation.
8. Procedure **f** calls **d** through the first closure in the continuation.
9. Procedure **d** returns a fail to **f** which relays the fail to **c**.
10. Procedure **c** has tried all of its alternatives therefore it returns the fail to **b** which returns to **a**, indicating that the query has failed all together.

Structure copying allows the control stack space of a determinate procedure to be reclaimed. In 'C' this translates to an immediate return to the calling procedure. Procedures **b**, **d**, **e**, **f**, and the last clause of **c** are all determinate; that is, they have no alternatives on backtracking, hence, there should be no need to return to them when failing. Consequently, these clauses should return without invoking any of the goals of the continuation list. To prove itself, the determinate clause passes only its local continuation (that defined by its body). For example, the second rule of the procedure **c** is determinate. Its first subgoal is, **f**, and there are no remaining goals, therefore, **f** should be passed an empty continuation list. When **f** returns, **c** has been proven, and it should return to **a**. The calling procedure, (e.g. **a**), must be able to recognize a determinate return, and invoke the next goal of the continuation itself. A PIC function, therefore, returns one of the following states to its caller: (1) success, which implies that all of the goals of the continuation passed to this procedure have been proven, (2) fail, or (3) determinate, which indicates to the calling procedure that a return is being made in order to reclaim call stack space. For this last case, the continuation goals have not been called. The example program now executes as shown in Figure 3.8 and as described below:

1. Procedure **a** calls **b** which immediately returns a determinate status to **a**. Therefore the call stack space of **b** is recovered.
2. **a** then calls **c** passing the tail of the continuation it had built for its call to **b**. The continuation built before the first call is the *working* continuation of this clause. The determinate return from **b** implied that **a** must itself invoke

Figure 3.8: Flow of procedure calls using continuation passing and determinate clause returns.

the next goal of the continuation, therefore the working continuation must be updated to reflect this call.

3. Procedure **c** calls **e** passing the received continuation; i.e. there are no goals to prepend to the list since **e** is the only subgoal.
4. **e** returns a determinate status.
5. Procedure **c**, which is still executing its first alternative, now calls **d** through the first closure of the continuation list, and passes the empty continuation tail.
6. Procedure **d** returns a fail status to its caller **c**. Note, however, that the failure relay through **b** has been avoided.
7. Procedure **c** moves to its second alternative. This is a determinate clause, therefore, only the continuation of this clause is to be passed to **f**. Since the clause has only one subgoal, the continuation passed is an empty list.

8. **f** returns immediately to **c**.
9. **c** has just completed a determinate clause, therefore, it also returns determinately to **a**.
10. Procedure **a** recognizes the determinate return, and calls the next procedure in its working continuation which is **d**.
11. Procedure **d** fails directly to **a** who concludes that the query has failed.

This addition to the control strategy has the advantage of limiting the call stack size. In the example above, the call stack space requirements were reduced by the size of two procedure environments. This is apparent when Figure 3.7, which goes to a depth of 5 calls, is compared to Figure 3.8, which goes to a depth of only 3 calls. However, this method introduces the need for non-local trail information. Without determinate returns, each function could keep track of its own instantiations in locally declared trail variables, therefore, the trail information would be distributed on the call stack. Because control does not return to all functions on backtracking with the updated scheme, this information must be passed between procedures to enable the choice point function to reset the state of the execution. Therefore, the trail stack, like the copy stack, will be dynamically created by the CPI function of Figure 3.1, and passed between PIC functions.

3.3 Features of PIC Clauses

So far, a simple clause, (one without cuts and ors), compiles into one case of a 'C' switch construct within the function which defines the corresponding procedure. This function has a calling sequence which includes: (1) a reference to a copy stack, (2) a reference to a trail stack, (3) a continuation list, and (4) the calling arguments. The stacks are dynamically allocated by the CPI function when the Prolog environment is created. This section will begin with an explanation of the PIC version of a simple clause. Cut clauses are then examined in conjunction with clause indexing since both define the determinacy of a clause. Finally, disjunctive and dynamic

clauses are added to the implementation, along with meta-evaluation, negative logic, and tail recursion optimization.

3.3.1 A Simple Clause

A simple clause in PIC format is organized in the following manner:

1. *Clause specific indexing instructions.*
2. *Instructions to get and unify the arguments with head goal terms.*

These statements have a WAM-like appearance, but will invoke other 'C' functions. If a failure occurs in this section, then the clause has failed, and the remaining statements must be ignored. The 'C' macro expansion facility, and the goto statement are used to reroute failures to a label where the execution is set to continue with the next alternative; for example:

```
GET_ATOM(calling sequence);
```

is expanded to:

```
if (get_atom(same calling sequence) == false ) goto failed;
```

This 'C' feature is used to reduce the amount of output the translator has to generate, and to provide a listing which is more consistent to the WAM compiler output.

3. *Instructions to put and unify the arguments of all the subgoal calls.*

These instructions construct the argument lists referenced by the closures of the local continuation.

4. *Build a clause specific local continuation list.*
5. *Make the next goal call.*

There are two cases for this group of instructions:

determinate clause — If this clause is classified as determinate by the translator, or by the clause indexing instructions, then the first subgoal is called and passed the tail of the local continuation since only the subgoals of this clause are to be proven before control is returned to the caller. If the clause is a fact, it immediately returns with a determinate status.

non-determinate clause — In this case, the local continuation list is prepended to the received list, and passed as a new, extended continuation when the first subgoal is called. If the clause is a fact, then the first goal of the received continuation list is called.

6. Respond to the returned status.

The action taken when control is regained depends on the status returned, and is one of:

failure — If a fail status is returned, then the current bindings are undone, and the next alternative is tried. If there are no alternatives in this procedure, then the fail is relayed through the return of this function.

success — If success is returned, then this function returns determinate or success according to the clause's own determinancy classification.

determinate — If determinate is returned, then for a non-empty continuation, the next goal is called, and the response is handled in the same manner as this first goal. For an empty continuation, the clause reacts in the same manner as for a success: i.e success or determinate is returned.

Steps 5 and 6 have been combined in the function **manage_action**. The local continuation is built to contain all of the subgoal closures, (including the first), and it is passed to this function according to the rules defined in step 5. Determinate returns are handled by the **manage_action** function which, therefore, returns only success or failure to the clause code. The code for this function, as it would stand for simple clauses, is given in Figure 3.9.

```

action manage_action ( continuation_list, stacks, and pointers )
contin      cont;
:
{
    action      next_action; /* action=(fail,success,determinate) */

    next_action = determinate;
    while (next_action == determinate) {
        if (cont == NIL)
            next_action = success;
        else {
            next_action = make the goal call defined by the first closure
                           of the continuation list;
            cont = tail of the continuation list;
        }
    }
    return(next_action);
}

```

Figure 3.9: Functionality of `manage_action` for a simple clause.

3.3.2 Clause Determinancy

The determinancy of a clause is recognized in two ways: (1) by the translator, PICG, and (2) by the clause indexing instructions. Both of these methods are approached in one of two modes offered by the translator, and selected by the user during the translation process.

Non-determinate Mode --- In this mode, the classification is restricted to only those clauses which are obviously determinate. For example, a clause whose only subgoal is the `!/0` predicate, is classified as determinate if the head goal is successfully unified with the calling arguments. This mode represents the standard Prolog model.

Simplified Determinate Mode — This assumes that the last alternative clause of any procedure is determinate.

The remainder of this section will focus on clauses classified as determinate according to the rules of the simplifying determinate mode.

Translator Recognition

Last Clause Determinancy — In determinate mode, the PICG always defines the last clause of a procedure as determinate. This implies that there is only one solution or that only one solution is required. This assumption is supported by: (1) experience, which showed that it is often the case, and (2) the significant decrease in memory requirements. An example of where this assumption is not true is in the following `generate_and_test` program:

```
generate_and_test :- do_it, fail.  
do_it :- generate(X), test(X).
```

This program should generate and test all possible situations found by the goal `generate/1` when `fail/0` invokes backtracking. The predicate `do_it` is not determinate because there are several solutions to `generate/1`. If the clause is assumed to be determinate then only the first situation is tested

before `do_it/0` returns to `generate_and_test/0` which subsequently fails. In order to make the last clause assumption valid, the program would have to be re-written such that the call to `fail/0` is contained in the same clause as the non-determinate goal `generate/1`. For instance the following program will execute correctly:

```
generate_and_test :- generate(X), test(X), fail.
```

Here, the clause will generate and test all situations before returning determinately to its caller.

Several papers have been written on the benefits of determinacy classification. In several cases the Prolog language is extended to allow the programmer to define the determinacy of each clause. This feature could be added to the PICG input, but it was deemed a cumbersome effort for the programmer [20, 14, 32, 27].

Cut Clauses - Any clause with a cut operator is defined as potentially determinate. If a cut clause succeeds, then it has no alternatives, and is therefore determinate by definition of the simplified mode. The implementation of a cut clause is slightly different from that of a simple clause. It is based on the idea that regardless of the success of the clause, the subgoals to the left of the cut do not need to know the continuation passed to their parent for the following reasons:

1. If these subgoals succeed, then the cut must remove any choice points defined during their proof. This is done automatically if the execution returns to the parent at this point.
2. If these subgoals fail, then the procedure moves to the next alternative clause, or returns a failure status.

The first subgoal of a cut clause is, therefore, passed a continuation which contains only the subgoals listed to the left of the cut. The parent clause regains control when the cut is reached, and can decide at that point, whether

```

    build a pre-cut local continuation list = b,c,d,nil;
    call manage_action passing to it this local continuation;
    IF manage_action returns a successful status THEN
        /* this clause is the last alternative */
        build an after-cut local continuation list = e,f,nil;
        call manage_action passing to it the new local continuation;
        IF manage_action returns a successful status THEN
            RETURN determinate;
        ELSE the clause has failed determinately
            therefore RETURN fail;
    ELSE continue with the next alternative in this procedure

```

Figure 3.10: Description of the control instructions for a cut clause.

to fail, or continue according to the selected mode as a determinate or non-determinate clause. For example, the clause:

```
a :- b,c,d,!,e,f.
```

would translate into the code given in Figure 3.10 if the determinate assumptions are to be made. For the non-determinate mode, the received continuation would be appended to the *after-cut local continuation* before the **manage_action** procedure is called, and either a **success** or **fail** status would be returned (cref. Figure 3.10).

This approach allows the call stack space, allocated to prove the subgoals to the left of the cut, to be reclaimed at the earliest possible time; i.e just before the **cut** is reached. It also eliminates any confusion over the ownership of the cut. If a **cut/0** function is called through a closure in the continuation list, then the parent and initial subgoals of the clause must be informed that their alternatives have been cut when backtracking occurs. The failure status which is returned to the cut procedure, must reach the grandparent goal, through a series of **return** instructions, without having taken any alternative branches.

This means that references to a subgoal's parent must be passed along in each goal call. If the cut function changes the relay of a failure status to something of the form, `cut(parent identifier)`, then the subgoals will know not to take any alternative paths. This, however, is further complicated when recursion is in progress. Here, a *recursion level* would also be required to determine which clause is being cut because all subgoals would recognize the same parent identifier. Therefore, `cut/0` will not be represented as a procedure; but rather, it will be handled by the parent clause as shown in Figure 3.10.

Clause Indexing

Last alternatives are also found using a technique based on the WAM clause indexing scheme where the type of the first argument is compared to the type of the first term in the clause, and the result is used reduce the clause search space. If the types are not compatible, then the the next alternative is tried immediately, otherwise the relative position of this clause is defined by the number of remaining alternatives which will match the type. There are several ways to code this task, but the version chosen is simple and gives every clause a consistent format. The implemented method only keeps track of how many clauses remain which will unify with a given type. In the WAM implementation, information is needed to determine exactly where the next alternative clause is. A `try_me_else` or `trust_me_else` instruction indicates where the code for the next applicable clause is found, and the `switch_on` instruction at the beginning of the procedure jumps to the first potential clause. PIC, however, simply determines if a particular clause can be used; i.e., each clause has its own set of selection instructions. If the comparison indicates that unification is impossible, then the next clause in the procedure is tried. The first set of instructions for this alternative are the clause indexing instructions, and they will determine if this clause is appropriate. The number of remaining clauses which will match a given type, defines the position of the clause. For example, if a clause represents a constant unification and none of the remaining do, then an incoming constant will cause this case to be classified as the last alternative.

An example, shown in Figures 3.11, 3.12, and 3.13, illustrates a predicate with

two clauses where the first argument of the first clause is the empty list constant, and for the second clause it is a list. This example contains the code which would appear if the translator is switched to determinate mode. The example shows that the first clause becomes determinate if the first argument is an atom, and that a **ref** (an unbound variable) is always unified. The second clause is always determinate because it is the last clause of the procedure. However, the indexing is still in place to avoid unnecessary unification of incompatible types. The example shows that the tag portion of the value cell is being used to select the case option, and that compound terms are being tagged with a specific functor/arity identifier. This will be discussed in Chapter 4. Impossible unifications cause the execution to jump to the **shallowfail** label shown in Figure 3.13. *Shallow backtracking* is a term used to describe backtracking within a procedure, *deep backtracking* is used when the state must be reset to a previous environment.

If, from a given point in the procedure definition, all remaining alternatives match the first argument with a variable term, then the clause indexing code is not generated because each clause must be explored. In this case, only successfully cut clauses and the last clause can be classified as determinate.

As in the previous discussions, the non-determinate version would append the received continuation to the local continuation before calling **manage_action**. When this call returns, a **success** or **fail** status is returned as opposed to **determinate** or **fail**.

3.3.3 The or Predicate

This section will define the translation of, and the control model to manage a disjunction in the PICG input. There are two types of statements containing disjunctions which are recognized by the translator.

Case 1

The simplest case arises when the **or/2** splits a rule into two unique clauses. In this situation, if the **or/2** predicate is written in prefix notation, then it will be the first operator in the clause body. For example:

```

{
  Determine arg1_type;
  while (continue==true)
  {
    switch (backtrack_level)
    {
      case 0 :
        switch (arg1_type)
        {
          case integer:  goto shallowfail;    break;
          case real:     goto shallowfail;    break;
          case atom:     determinate = true;  break;
          case ref:      determinate = false; break;
          default:       goto shallowfail;    break;
        }
        GET_ATOM(nil_0, arg_list element, ...);
        continue with gets and get/unif pairs;
        perform the puts and put/unify pairs;
        set up the local continuation
        call manage_action;
        break;
    }
  }
}

```

Figure 3.11: Psuedo-code to illustrate clause indexing - Clause 1.

```

case 1 :
    switch (arg1_type)
    {
        case integer: goto shallowfail;    break;
        case real:    goto shallowfail;    break;
        case atom:    goto shallowfail;    break;
        case ref:      determinate = true; break;
        default:       if (arg1_type = list_2)
                        determinate = true;
                        else goto shallowfail;

        break;
    }
    GET_LIST(arg_list element, ...);
    continue with gets and get/unify pairs;
    : as in case 0
    break;
default:
    continue = false;
    status = fail;
    break;
} /* end switch(backtrack_level) */

```

Figure 3.12: Psuedo-code to illustrate clause indexing - Clause 2.

```

switch (status returned by manage_action or set by the default case)
{
    case fail:  if (determinate == true)
                  continue = false;
                else { clean up the stacks ;
                      ++backtrack_level; break; }
    case success: continue = false;
                  if (determinate == true)
                      s+atus = determinate;
                  break;
}
goto loop;

failed:  if (determinate == true)
          continue = false;
        else { clean up the stacks
              ++backtrack_level; break; }
shallowfail:
          ++backtrack_level;

loop:  ;
} /* end while */
return(status);
} /  end procedure */

```

Figure 3.13: Psuedo-code to illustrate clause indexing - Control Code.

1. $a :- (b,c,d;e,f),g.$ *is equivalent to*
2. $a :- ;((b,c,d),(e,f)),g.$ *and*
3. $a :- b,c,d,g.$
 $a :- e,f,g.$

In this example the third definition of $a/0$ consists of two clauses whose computations do not overlap prior to the disjunctive goals. Therefore, regardless of the presentation, (as one clause with a disjunction, or as two clauses), and the outcome, the same amount of processing is required to prove the clause. The translator recognizes these situations, and creates two alternatives in the output function.

Case 2

All other clauses containing an **or/2** predicate fall into this second category; i.e., the **or/2** splits the clause after a set of common subgoals. For instance the clause:

1. $a :- b,c,(d;e,f),g.$ *is equivalent to*
2. $a :- b,c,d,g.$
 $a :- b,c,e,f,g.$

If the proof fails during the call to $d/0$, and the second representation is being used, then the common subgoals listed before the **or/2** predicate, $b/0$ and $c/0$, would have to be re-evaluated. This effort is avoided in the PIC implementation by extending the continuation list idea to represent a disjunction, and the **manage_action** procedure to process a disjunctive continuation.

Unlike the cut operator, the choice point alternatives introduced before the **or/2** operator are viable paths in the execution, therefore, control cannot be returned to the parent's code when the **or/2** is reached. The continuation must reflect the options before the first subgoal call is made. Previously the list elements of a continuation list were defined to contain a closure and a pointer to the next element. This is extended to accommodate the disjunctions. A type identifier (**and,or**) is added and accompanied by a corresponding data structure. An **or** type identifier

implies that the proof has reached a disjunction of goals. Continuation list elements of this type contain three continuation pointers: (1) a pointer to the continuation list which represents the first alternative of the **or/2**, (2) a pointer to the continuation list which defines the second alternative of the **or/2**, and (3) a pointer to the continuation list which represents the rest of the proof. Figure 3.14 illustrates a continuation list which contains an **or** element. This list would be built as the local continuation for the clause, **c :- b, (c;d), e**. For a non-determinate clause, the received continuation is appended to the end of the remainder list before **manage_action** is called to process the list.

The **manage_action** function, given in Appendix B, tests the type of a continuation element before invoking the next goal's procedure. If an **or** element is reached, two new continuation lists are created. The first is a reference through the **alternative 1** pointer of the **or** element, and the second through the **alternative 2** pointer. The **remainder** goals are appended to the end of each new list. If the last element of the list is of type **and**, then the **next** pointer will be **NIL**. If the last element is of type **or**, then the **remainder** pointer will be **NIL**³. To continue with the proof, the **manage_action** function is recursively called to prove the first alternative path. If this returns a fail status, the function calls itself again, passing the second alternative's continuation list. The result of this call represents the status of the parent clause. Essentially, the **or/2** predicate has been implemented with the Prolog definition of § 2.1.4 which was:

```
or(X,_) :- X.
or(_,Y) :- Y.
```

A cut nested in an **or/2** operator, (e.g. **a:-b, (c,d,!;e)**) represents a problem for the current design. A cut predicate, which is not nested, is being handled by the code of the parent clause. However, if it is to be called from within an **or/2**, then it must be represented in a continuation list, and called using a closure. An earlier discussion explained the complications of having a **cut** procedure. If a new return status, **cut**, is added and returned by the **cut/0** predicate, parent identifiers

³ The clause **a:-b,c, (d;e)** would require a **NIL** remainder pointer.

c :- b, (c;d), e would create the following local continuation

Figure 3.14: Example of a continuation containing **or** calls.

and recursion levels would have to be included with each call in order to relay the failure to the correct clause. This extra overhead does not seem to be worth the gain acquired by allowing cuts in an **or/2** structure, therefore, it was decided that all cuts must occur in the top-level of a clause's subgoals. Cuts within the **or/2** predicate of a clause with the form described as case 1 are tolerable because the separation is made very early on in the translation.

The problems which arise from the **cut/0** operator result from using the 'C' call stack for the control stack in the PIC execution. This forces the strict ordering of environments on the stack. The current environment must always be at the top of the stack, and environments can only be removed by a return instruction within the procedure. Unlike the WAM design, choice point environments cannot exist before backtracking to the corresponding code, therefore, alternatives cannot be invoked by a simple code pointer update. This lack of flexibility and information complicates finding the correct alternative.

3.3.4 Dynamic Clauses

Dynamic assertions in the interpreter model of Chapter 2 require run-time creations of prototypes which are referenced through the atom table. The statement:

```
assert(mother(betty,cathy))
```

shows how the built-in predicate **assert/1** is typically used in a Prolog program. The clause to be added, **mother(betty,cathy)** is an argument of the **assert/1** goal, and is therefore built as a structure on the copy stack before the call is made. From this structure, the atom table reference **mother/2** and the prototype are defined. The reference indicates to which procedure list the clause is to be added, or for a clause retraction, the procedure list from which the element is to be removed.

In the PIC execution model, prototypes are not being used for clause representation. Dynamic assertions of PIC would require run-time code creation, compiling and linking where the code creation includes the translation of the Prolog statement to PIC. To implement this, a 'C' version of the PICG would have to exist in every target program strictly to manage the dynamic clauses. However, if dynamic clauses

are restricted in the same way as in MProlog, then they can be stored as prototypes as described above. MProlog requires that dynamic predicates be:

1. unique, and designated (specifically declared) predicates, (ones which have no alternatives when the program begins and therefore no function exists with their name), or
2. dynamically declared within the `create_dynamic/1` predicate which deletes any previous definition of the procedure⁴.

With these two options, it appears that dynamic predicates are treated differently in MProlog also. In PIC, the prototypes will have the same form as the copy stack structure, and can, therefore, be passed as arguments to the existing unify functions.

The declaration of a dynamic clause must appear at the beginning of the Prolog program. For instance:

```
dynamic(dynamic_clause_1/3).
```

When a pre-declared dynamic predicate is encountered in the original program, e.g:

```
num_rhats(Y) :- dynamic_clause_1(hats,Y,red).
```

it is replaced with:

```
num_rhats(Y) :- spy_dynamic(dynamic_clause_1(hats,Y,red)).
```

This way, the clause is converted into an argument and passed as a structure on the copy stack to the `spy_dynamic/1` procedure⁵. This procedure unifies this argument, (the clause), with the asserted structures. An example of a corresponding assertion is:

```
add_statement(dynamic_clause_1(X,6,red)).
```

⁴This has not been implemented, but poses no problems.

⁵It is interesting to note that MProlog offers the analogous `matching_statement/1` predicate

Fact Assertions

To assert a fact, a copy of the terms is made in a dynamically allocated mini copy stack in the *dynamic clause space*. The copy defines the clause prototype, and it is referenced through a field in a linked list element of the procedure definition (cref. § 2.2.1). The procedure is referenced through the atom table as in the interpreter model. When a dynamic assertion is made, the terms are copied, not unified, therefore, there are no bindings between the asserted clause and the argument of the `add_statement/1` goal. If the fact was ground when it was asserted, the `spy_dynamic/1` predicate unifies the terms of it's argument with the value cells of the asserted mini stacks. Otherwise, an instance of the clause must be created by making a copy of the prototype on the copy stack before unification can begin. As in the interpreter implementation, each remaining element in the list of asserted clauses, represents another alternative in the proof, therefore, the `spy_dynamic/1` function tries each clause until a match is found, or all have been explored.

Rule Assertions

Dynamic rules introduce several new complications. The software required to manage dynamic rules has not been implemented, but an approach is provided in this section. Zero arity dynamic rules are an exception. This category of rules can be asserted as facts whose arguments are a conjunction of subgoals. These goals can be evaluated using meta evaluation (discussed in the next section). For example:

```
add_rule :- add_statement(rule(goal1(X),goal2(X,Y),goal3(Z))).
evaluate_rule :- rule(X), evaluate(X).
```

A rule is built as an `'-/2'` structure when passed as an argument in a goal call. The first element of this compound term is the head goal, and the second element is the body. The head goal should be asserted in the same manner as a fact: as a stack of terms which allow the existing unification functions to be used. The body represents an evaluable conjunction of goals which must be transformed into a continuation list to be compatible with the implementation model. A rule requires two "prototype" pointers: (1) a pointer to reference a mini stack definition of the

head goal, (exactly as was done for a fact), and (2) a pointer to the continuation of the rule. When the `spy_dynamic/1` procedure is called and passed a rule, the terms of the head are unified with the head prototype, argument registers are defined for each subgoal, and the `manage_action` procedure is called and passed the rule's continuation appended with the continuation received by `spy_dynamic/1`. Meta evaluation is an alternative approach to evaluating the body of the rule. In this case, the body of the clause is stored, like the head, as a prototype (a copy of the argument passed in the call).

Cuts predicates are not allowed as subgoals of a dynamic clause because of the problems involved in determining the parent of a cut when the goal is invoked through an entry of a continuation list (cfr. the cut clauses section of § 3.3.2). However, `ors`, `nots`, and declared dynamic goals can be managed. In general, dynamic clause assertions, especially rules, are quite inefficient because of the run time memory allocations, and the amount of copying required to complete the work.

3.3.5 Meta-evaluation and Negative Logic

Meta-evaluation allows the programmer to pass a Prolog expression as data, and have it evaluated. The built-in function `evaluate/1` is used for this purpose. It is defined in PIC as:

```
evaluate((A,B)) :- !, evaluate(A), evaluate(B).
evaluate((A;B)) :- !, (evaluate(A) ; evaluate(B)).
evaluate(A) :- meta_evaluate(A).
```

The function `meta_evaluate` can be expressed in Prolog as:

```
meta_evaluate(A) :- A.
```

In the PIC implementation, data and program are not stored in the same manner; that is, data is stored as a value cell and prototype, and program is stored as code. To provide this feature, each function identifier is hashed into the atom table, where the corresponding function pointer can be retrieved. The argument passed to `meta_evaluate` is either an atom, (for a zero arity function), or a compound

term. For an atom, the value pointer is used to retrieve the function pointer, and no arguments are sent in the call. For a compound term, the structure pointer indicates the function to be invoked, and the arguments are found immediately above this tag on the copy stack. A set of argument registers are, therefore, allocated and defined as reference cells to the terms of the structure on the copy stack.

Having defined **evaluate/1** and **meta_evaluate/1**, negative logic is a simple addition. The built-in function **not/1** is provided and defined as:

```
not(X) :- evaluate(X), !, fail.
not(X).
```

If the evaluation of **X** succeeds, then **not/1** returns a fail status. Otherwise, the second alternative clause is used, and the execution state is reset to the that which existed on entry into the **not/1** procedure. Regardless of the success, the copy and trail stacks do not contain any bindings which resulted from the evaluation of **X**. This is useful for saving space when these bindings are not of interest, but the success of the evaluation is important. In this case, the common approach taken by Prolog programmers is to nest one call to **not/1** within another; e.g. **not not Y**.

3.3.6 Tail Recursion Optimization

Tail recursion optimization (cref. § A.5), was described as a method for saving control stack space with iterative clauses. Last call recursive clauses with no choice points between the parent and the last goal can reuse the same environments without losing any information, therefore an N-fold recursive rule, uses 1 instead of N sets of environments for the clause's proof. The translator recognizes only two, brute force, situations where TRO might be exploited: (1) in clauses which include a cut immediately before the recursive call, and (2) in single subgoal recursive rules. The determinancy of the clause is evaluated by the translator, or by using clause indexing as discussed earlier in this section. When a clause is classified as determinate and tail recursive, the execution continues by: (1) resetting the **backtrack_level** to zero, and (2) assigning the arguments to be passed in the recursive call to the argument

parameters passed to this procedure⁶.

The savings in call stack space is somewhat offset by extra coding required to copy the argument value cells from one place to another. The extent of copying is complicated because of the following situation:

If the proof fails after successfully proving the TRO'ed goal, and backtracks to a choice point between this goal and the parent, then the original argument parameters are no longer intact. This is because they were over-written by the tail recursive clause, and have not been reset by the parent goal. For instance, given the clause:

```
a(LIST):- goal_with_several_alternatives(LIST),
          member(ELEM,LIST),
          fail.
```

when `member` has exhausted all of its alternatives, the fail will return to `goal_with_several_alternatives/1`. Eventually, `member/2` will be called again. The contents of the argument list passed in this call will reflect the last loop of tail recursion performed in the previous execution of this goal.

To correct this, the original arguments must be saved when TRO begins, and re-stored when backtracking through this procedure.

⁶As an example, the PIC version of the `member/2` (cref. § A 5) is explained in detail in Appendix C.

Chapter 4

Unification in PIC

Chapter 3 defined several requirements for the PIC translator and its output. Those which directly affect the ‘C’—Prolog interface will be considered in this chapter while developing a supporting method for term representation. Henceforth, specifics on WAM-like variable classification and unification are explained.

4.1 ‘C’—Prolog Interface

The ‘C’ procedures in the mainline of the target system must not be concerned with Prolog specific requirements like the stacks and term representation. The CPI functions have been introduced to provide a smooth and simple interface between the two programming techniques, and it is the job of the CPIG to produce these functions.

The CPIG, (the CPI generator of Figure 3.1), precompiles a ‘C’—Prolog call into a corresponding, boolean CPI function. This function creates the required Prolog environment, and returns the requested instantiations to the ‘C’ environment. The preparations include: (1) the dynamic allocation and initialization of a copy stack, a trail stack, and the global pointers associated with them, (2) the transformation of ‘C’ parameters into Prolog terms, (i.e., the construction of value cells and copy stack structures), (3) the initialization of an empty continuation list, and (4) the invocation of the appropriate PIC function. Before returning to the ‘C’ caller, the

CPI function must: (1) assign the requested instantiations to their corresponding 'C' structures, and (2) convert a **fail** to a **false** return, and either **success** or **determinate** to a **true** return.

The shared (passed) variables pose two problems.

1. How can the variable type be determined for term tagging?
2. How can the mode of the variable (input or output) be determined?

The CPI generator must be able to resolve these questions through the standard and user defined type declarations, and user defined function declarations in the PICDEF and CPIDEF files. The type declarations help resolve the first problem, and the both problems are mastered by the function declarations, which are extended to include type and mode information for each argument of a procedure. For example, a 'C' integer with the value zero would translate to the following value cell if declared to be an input parameter: `<int,0>`. As an output variable, the current value is ignored, and the Prolog representation would have to reflect that the value is yet undefined, therefore, `<undef,undef>` is a potential, unambiguous representation.

As an example of what the user is required to provide in the CPIDEF file, the `member/2` predicate, which is repeated in Figure 4.1, is considered.

1. The structure definition for the list in the call must be declared; e.g. a list of integers would require the following declarations:

```
struct intlist {
    int          term;
    struct intlist *nxt;
}

typedef intlist *int_list;
```

A `*nxt` pointer is the convention chosen to indicate a nested structure of the same type (e.g. the tail of a list is a list). Here, a `cons/2` structure is being


```

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

```

Figure 4.1: **member/2** predicate

defined. It will be called **list/2** in this implementation to emphasize the importance of the imperative view.

2. The **member/2** PIC function must be declared with the type and mode of each argument as it will occur in the interface call. The syntax expected in place of each argument is *type:mode*, where the mode is either **in** or **out**; e.g. **int:in**. The PIC functions will not assume any type or mode restrictions, therefore, within the Prolog environments, the calls to **member/2** can be made with any type of list structure. The declarations pose limitations only to the 'C' to Prolog interface. Continuing with the example started above, the 'C' programmer could declare **member2** as a function which determines a member of an integer list as in:

```

member2( int:out, int_list:in );

```

3. The 'C' variables to be passed as arguments must be declared in the program segment; e.g.:

```

struct int_list testlist;
int          element;

```

4. The call in the 'C' code segment would be presented as:

```

member2( element, testlist );

```

This call would expand into a set of instructions which: (1) create an unbound instance of **element** as the first argument in the call, and (2) construct a copy stack representation of **testlist** which repeatedly uses the **list/2** tag, and

terminates with the `nil` atom, and which is referenced through the second argument of the call.

The second requirement listed above indicated that a function declaration must be made for the Prolog procedure in the sense that it will be used in a 'C' Prolog call. A desirable extension would be to allow the 'C' programmer to define several usages of the same Prolog predicate. For instance, the `member/2` procedure could be declared alternatively as:

```
member2( int:in, int_list:in );
```

This definition would return a `true` status if the integer passed in the first argument is an element of the list passed in the second argument. The CPIC would require an indication that this is a different declaration for the same Prolog function, and it would have to be able to distinguish one call from another in the 'C' program. A possible syntax adds a version number with the declaration and call. For example, the declarations might include a *_version number*, i.e.:

```
member2_1( int:out, int_list:in );
member2_2( int:in, int_list:in );
```

The corresponding calls would be:

```
member2_1( element, testlist );
member2_2( element, testlist );
```

In the first call, the `element` parameter is converted to an unbound term before the PIC function is called. In the second call, this parameter is passed as an integer with the current value (e.g. `<int,0>`).

The *primitive*, 'C' types which can be defined in a 'C' Prolog call are:

1. `string` - for a character string which is to be managed as an atom in the Prolog environment, therefore, the character string is hashed into the atom table, and referenced and unified through a pointer.

2

MicroD

2. `quoted` - for a character string left as is in the Prolog environment. This type is required for character strings which cannot also be used as 'C' identifiers. For example the string `"apple"` can be accessed with the identifier `apple.0`, however, a string which contains a blank space does not have a simple matching identifier. For this type, unification might require a string comparison, which is a time consuming effort, however, it is expected to be used mainly with the output functions where they will be bound to a variable and printed as is.
3. `int` and `int*` - for integers.
4. `float` and `float*` - for floating point numbers.
5. `void` - for an initially uninstantiated (output) variable. These parameters are left in their Prolog representation on return.
6. any structure containing a primitive type and any number of `*nxt` pointers. A single `*nxt` pointer defines a list of a primitive type; i.e., a "simple" `list/2` structure.

All compound terms, including a non-primitive structure of a list element, must be separately and fully declared by the user in such a way that: (1) the functor/arity can be determined, and (2) that each term can be decomposed into a primitive, or structures of primitive types. For example, if the Prolog code expects a `stuff/3` structure, the translator sets up the code for `stuff.3` to be an entry in the atom table. When this structure is being constructed on the copy stack, the atom table reference for `stuff.3` marks the beginning of the structure, and it is followed by the value cells for each of the three arguments (§ 2.2.2). Because all structures must reside on the copy stack, a 'C' variable which is to be passed as this type must also be transformed into a copy stack instance which is referenced by a `<ref, pointer to copy stack>` value cell in the calling arguments. If `stuff/3` is of the form:

`stuff(integer,quoted,simple list of strings)`

the CPI generator will require the following declarations:

1. A declaration of a list of strings; i.e.:

```

struct strlist {
    string      str;
    struct strlist *nxt;
}

typedef strlist *str_list;

```

2. A declaration of the type `stuff_3`; i.e.:

```

struct stuff {
    int      *t1;
    quoted   t2;
    str_list t3;
}

typedef stuff *stuff_3;

```

3. A declaration of the parameter to be passed in the 'C'—Prolog call:

```

stuff_3      param1;

```

The CPI generator is provided a set of primitive term *construction* procedures which return a pointer to a value cell created at an indicated location; i.e., in an argument register, or on the copy stack. For a compound term the argument register is assigned a **ref** tag and a pointer to the copy stack. The generator also has access to a set of primitive term *decomposition* procedures which return a value or string. The type declarations provided in the CPIDEF file are converted into a set of instructions which transform a 'C' parameter into a Prolog term and vice versa. The calls are set up so that a value, a pointer, or a pointer address can be passed in the 'C'—Prolog call. The CPI generator is responsible for recognizing what is required to be able to copy and pass input parameter values to the PIC functions,

and assign instantiations to the appropriate 'C' locations on return. The generator is converting the representation supplied by the user to extract the information it needs. It should be noted that pointers do not exist in the Prolog language.

To assign a value to a 'C' structure element as a result of a 'C' Prolog call, the structure must be defined as an input parameter, and the specific term must be an output parameter. For example, to return the integer portion of a `stuff 3` structure, the function declaration defined by the 'C' programmer in the C'IDEF file might be:

```
get_stuff( int*:out, stuff_3:in );
```

The call would be:

```
get_stuff( param1->t1, param1 );
```

And an example Prolog application translated by the PICG is:

```
get_stuff(X,stuff(X,"from garage",STUFF)) :-
    accum_stuff(X,garage,STUFF).
accum_stuff(X,garage,[Y|T]) :-
    retract(garage_inventory(Y)),
    accum_stuff(NUM,garage,T),
    X is NUM + 1.
accum_stuff(0,Z,[]).
```

The extra information is required to cause the value to be returned; i.e., in this example, the 'C' routine is not returned the list of items found in the garage even though it is being generated. Only those arguments which are defined to be outputs are returned.

Where Warren uses a reference cell which points to itself to indicate an unbound variable, this implementation uses a `NULL` value pointer to indicate an uninstantiated variable. In the PIC implementation, a term can be *type bound* without being *value bound*, however, these terms will always be part of the 'C' Prolog interface. For instance, an argument defined to be `int:out` would be transformed to `<int,NULL>`

Figure 4.2: The PIC value cell *mini stack* for `<integer,3>`.

as opposed to `<undef,NULL>`. The type assigned by the 'C' declaration of the variable, is considered throughout the Prolog proof regardless of whether the parameter is an input or output argument.

4.2 Data Representation

From the specification of the interface in the previous section, the implementation of a PIC value cell is defined to be a two field data structure whose first element is a tag, and whose second element is the value pointer. The tag field contains a pointer into the atom table at the entry for the corresponding term type (e.g. `ref` would be an atom table entry). The value pointer is `NULL` if the term is unbound, otherwise it is an integer, float, character, reference, or atom table pointer. The data structure format is that of a stack of `obj_ptrs`. This coincides with the form required for the copy stack implementation which must also be a stack of pointers. An example value cell is illustrated in Figure 4.2. In the example, `^ integer`, refers to a pointer to the atom table entry for `integer`, and, `^ 3`, is a pointer to the value 3.

WAM introduced the idea of determining term types, and classifying variables as temporary, permanent, or unsafe at compile time to simplify unification, and save physical execution space respectively (cref. § 2.4.1). The typing, classifications, and storage of terms in the PIC implementation are presented in this section.

4.2.1 Data Typing

The PIC translator detects and distinguishes between the following term types: integers, reals, strings (atoms), quoted strings (character strings with at least one

```

qsort([],R,R).
qsort([X|L],R0,R) :-
    split(L,X,L1,L2),
    qsort(L1,R0,[X|R1]),
    qsort(L2,R1,R).

```

Figure 4.3: `qsort/3` predicate.

space character), variables, and structures which are each denoted by their specific functor/arity pair.

4.2.2 Variable Classifications

Within the variable type, each term is further classified as temporary, permanent, or unsafe according to the same rules used by WAM. A short summary is repeated here:

Permanent — A permanent variable is one which occurs in more than one goal of a clause (where the head and first goal together are counted as one).

Unsafe — An unsafe variable is a permanent variable which did not first occur in the head goal or within a structure. The bindings of these variables disappear when the parent's environment is reclaimed on a determinate exit. They are called unsafe because they can be referenced by variables which have been introduced since themselves, and these bindings are lost when a determinate return occurs. If necessary, an unsafe variable is "globalized" (put on the copy stack) to avoid such dangling references.

Temporary — All remaining variables are temporary.

For example, the `qsort/3` predicate repeated in Figure 4.3 would have its terms typed and classified in the following fashion by the PIC translator.

- In the first clause, the empty list symbol is recognized as an atom, therefore it is of type string with value `nil_0`, both of which are implemented as pointers to atom table entries.
- `R` is a variable whose first occurrence is typed as a `var`, and all remaining occurrences as `value`: vars use the `get_var` or `put_var` instructions and values use `get_value` or `put_value`; (§ 2.4.2).
- The variable, `R`, is classified as temporary because it occurs only in the head goal.
- In the second clause the head goal arguments are typed as follows: (1) a `list/2` structure whose first and second elements are `var` type variables, (2) a `var` variable, and (3) a `var` variable. The variables are classified as: `X` is permanent because it occurs in three goals, `L` is temporary because it is found only in the head and first goal, and `R0` and `R` are permanent because they both occur in more than one goal.
- For this clause, all other occurrences of the variables typed in the previous step as `var`, are typed as `value`, and they are assigned the same classification and storage location.
- Variables `L1` and `L2` are typed in the same manner, but are classified as unsafe. Their permanency is defined by the fact that both reside in more than one goal, but they are unsafe because their first occurrence was neither in the head goal nor in a structure.
- The implementation also recognizes variables whose first and only occurrence is in the head, and thus do not require any step for unification. These are typed as `void`, and from that point on, are only dealt with by the portion of the translator which generates the instructions of the clause indexing scheme.

4.2.3 Data Storage

In WAM, each subgoal returns to its parent when that goal has been proven through a *parent pointer* in their environment. The parent regains control of the execution before the next subgoal call, and can assign the appropriate values to the argument registers immediately before each call (see Figure A.7). Temporary variables are stored in the extra argument registers of each call, and they exist only during that goal's proof. However, the PIC implementation must define all subgoal arguments before the first goal call is made, and, therefore, must allocate as many sets of argument registers as there are goals in the rule (cref. § 3.3). Permanent variables are allocated control stack locations, and exist as long as the environment exists in both WAM and PIC, but the optimization techniques of WAM are not available to the other as follows.

Arguments and Temporary Variables

The PIC implementation allocates the arguments registers as locally declared mini (copy) stacks which means that these lists are part of the activation record of a procedure. In this form, the unification functions do not have to differentiate between an argument and a copy stack term, thus trailing, binding, and dereferencing are performed strictly on stacks of value cells. However, temporary variables, which are stored in the spare entries of the received argument list, exists as long as the parent has not returned. The maximum number of subgoal calls for a clause in a given procedure defines the number of argument lists which are declared. For simplicity, the lists are a fixed size and are statically declared since locations are removed automatically by the **return** instruction. For consistency, temporary variables are stored in the extra locations of the procedure's argument stack. This offers some space savings in that existing locations are being used instead of creating new ones, but it does not represent true "temporary" locations.

Permanent Variables

Permanent variables are managed, as in WAM, on the control stack (the ‘C’ call stack) by declaring local variables. Each variable is defined to be a value cell stack: i.e., a tag followed by value as shown in Figure 4.2. Unfortunately, Warren’s optimization techniques, which place the shortest lived variables at the top of the environment so that they can be removed as soon as possible, do not provide any advantages in this implementation because environment trimming isn’t available to a PIC function. In ‘C’, the size of the environment (activation record) is fixed, and allocated when the call is made, therefore, local variable locations cannot be deallocated by the code. Besides, the parent does not regain control of the execution before each subgoal call, and this is when trimming would take place if it could.

Unsafe Variables

Unsafe variables cannot be easily “globalized” when their last occurrence is referenced because the parent does not necessarily regain control of the execution to test for potential problems. The continuation passing control strategy has again removed the opportunity to optimize the data management. Unsafe variables are globalized on their first occurrence as if they are compound terms. This implies that each unsafe variable is constructed as an unbound variable on the copy stack, and referenced by the permanent variable location assigned to that variable.

4.3 Unification

Unification is performed through the **get**, **put**, and **unify** instructions generated according to the type and variable classifications discussed above. The instructions used in WAM were listed in § 2.4.2. For the most part these are unchanged with respect to when they are used and the overall result of their invocation. The differences are listed in § B.2. This section will describe the details of the three main components of unification: dereferencing, binding, and trailing.

4.3.1 Dereferencing

Dereferencing is performed by following the value pointers of a series of reference cells until a non-reference cell or an unbound variable is reached. Pseudo code for the **dereference** function used in the PIC implementation is provided in Figure 4.1.

This procedure continues until the first **NULL** value pointer is reached; i.e., it stops when the first type bound or unbound variable is found. 'C' variables are the only potentially type bound, but not value bound terms. They are also, by default, the most senior terms in the execution, therefore, this is a suitable stopping condition. Every value cell pointer, (pointer to an argument stack, the copy stack, or a permanent variable location), is dereferenced early in a **get** instruction's functional expansion. The one exception is the **get_var** instruction which simply copies the contents of the argument cell into the local variable. It should be noted that the **unify** instructions often use the corresponding **get** instruction to complete the unification of the structure arguments when they are executing in read mode. Bindings and trailings are performed on fully dereferenced value cells; i.e., the most senior representation of a variable.

4.3.2 Binding

The dereferencing function returns a status for the term based on the contents of the tag. A fully dereferenced term which is tagged as a reference cell represents an unbound variable and it is indicated as such by this returned parameter. All other variables are said to be bound, however, their value pointer may be **NULL** as is the case of a type bound, output mode, 'C' variable. With this information, the general purpose unification function, which is called by **get_value** and indirectly by **unify_value** when in read mode, calls one of three specialized binding functions to complete the instantiations and the trailing. These are: **bind** for binding one unbound variable to a minimally type bound variable, **unify_refs** for binding two uninstantiated variables according to their relative seniorities, and **unify_values** for matching two minimally type bound variables. Except for calls to the trailing functions, the remaining **get** and **put** instructions are self-contained. They represent

input parameter: *pointer to a value cell*
which is type obj_ptr in code

output parameters:

- (1) *pointer to dereferenced value cell*
- (2) *status of the type binding:*
true - if term has a specific type,
false - if unbound variable.

start with local variable found = false, and
working pointer = input pointer.

```

WHILE (found == false) DO
  IF (pointing to a reference cell) THEN
    IF (the value pointer is NULL) THEN
      binding status = false;
      found = true;
    ELSE
      assign the value pointer to the
      working pointer;
    ENDIF
  ELSE
    binding status = true;
    found = true;
  ENDIF
ENDWHILE
RETURN(working pointer)

```

Figure 4.4: Dereferencing Algorithm

specific cases of unification and perform the necessary matching or assignments required to satisfy their part of the general purpose algorithm.

4.3.3 Trailing

All instantiations to reference cells which are not on the copy stack are trailed. For the copy stack, a choice point is defined and passed between the PIC functions; i.e., the calling sequence is extended to include the nearest choice point of the copy stack. Only those bindings which are made before this point need to be trailed. The remaining instantiations are automatically removed when a clean up function (called when a failure occurs) resets the top of the copy stack pointer to this choice point. The nearest choice point is evaluated by each PIC function clause as follows:

Non-Determinate Clauses — A non-determinate clause has alternatives, therefore, the nearest choice point passed to the **manage_action** function is updated to the current top of the copy stack. This update is made before this clause is unified with the arguments because the choice point is the top of the copy stack when this procedure was invoked.

Determinate Clauses — This refers to any clause which does not have alternatives; i.e., one which is determinate under the simplifying assumptions (cref. § 3.3.2). The choice point which was received by the parent procedure is passed on in the next goal call.

When the execution backtracks, the copy stack is cleaned up just before the **backtrack.level** is incremented in the procedure with the nearest choice point (cref. Figure 3.13). The memory which is dynamically allocated when a quoted, integer, or real term is created is not being returned to the free memory pool during the *clean up* phase. This exercise was considered to be too time consuming to benefit most applications because the tagged type of every value cell referenced by the trail stack entries, must be examined before the deallocation can be performed.

4.4 Example

Appendix C contains the translator, generated output of the `member2` predicate given in Figure 4.1. Along with the code is a statement-by-statement description of the execution for the ‘C’-Prolog call equivalent to `member2(3,[1,3,5])`. The example gives an overall picture of a PIC function’s execution, and in particular, it highlights the representation of the variables in the appropriate data structures as created by the CPI function and the Prolog environment.

Chapter 5

Optimizations

5.1 Tuning the PICG Output

Before putting a target system with PIC code into practical use, it is recommended that the output be tuned to some degree. Optimizations to the translator output are based on information known by the programmer which is not made available to the translator. Knowledge as simple as how many subgoals are in a clause can be used to save space on the call stack, while detailed schemes to recognize determinancy can be used to achieve a more noticeable savings in the same regard. Other techniques might be applied after learning how the program will use the PIC procedures. For example, mode information such as that required by the CPG (cref. § 4.1) can be used to reduce the efforts of unification, or increase the efficiency of clause indexing. This section discusses the techniques which can be used to optimize a PIC procedure in terms of: (1) the knowledge required for the update, and (2) the update itself. The enhancements are divided into three categories according to the type of information needed with respect to a clause or procedure. These are:

1. simple adjustments which are based on the clause definitions,
2. updates due to argument modes and types, and
3. updates due to clause or procedure determinancy.


```

action unify2(arg_list, continuation, global_stack,
              trail_stack, choice_point)
arg_list_type    arg_list;
contin           continuation;
g_stack          global_stack;
t_stack          trail_stack;
int              choice_point;
{
  obj_ptr         *deref1, *deref2;
  boolean         bound1, bound2, status;
  deref1 = dereference(&arg_list[0], &bound1);
  deref2 = dereference(&arg_list[2], &bound2);
  status = unify(deref1, deref2, global_stack,
                trail_stack, choice_point);
  if (status == true)
    return(success);
  else
    return(fail);
}

```

Figure 5.7: Hand-compiled version of the unify/2, =/2, predicate.

declarations, then even further refinements can be made¹. The improvements are essentially the result of being able to specialize the unification instructions to a direct assignment or comparison; i.e., they add one more step to the specializations determined in Warren's compilation techniques. In this discussion modes will be represented by:

- + for an input argument,*
- for an output argument, and*
- ? for an argument whose mode can be either input or output.*

The specific argument modes of a procedure are defined by writing the predicate functor followed by a mode indicator in place of each argument. For instance, if the **member/2** predicate of Figure 4.1 is always expected to be passed a list in the second argument, but is expected to be used to both generate members of the list and to test an element for membership, then its mode definition would be represented by the following: **member(?,+)**: i.e., the mode of the first argument can be either way, but the second argument is always a complete list structure². The updates which can be made from mode information are listed below in order of increasing complexity:

Mode testing — If a variable is defined to be an input argument, then it must dereference to a value cell with a tag other than the **ref** tag. If a variable is defined to be an output argument, then it must dereference to a value cell with a **ref** tag. The status of the instantiation of an argument is returned in the second argument passed in a call to the **dereference** function (cref. § 4.3.1). This type of test would be performed before the **switch** block which cases each clause alternative, and if the test failed, then the procedure could fail immediately.

Type testing — If a particular type is expected on an input argument, then it must dereference to a value cell with the corresponding type. This test would be

¹The advantages mentioned here are in efficiency, not flexibility or maintenance.

²"Complete" implies that the list is terminated with a **nil** atom as opposed to an unbound variable.

performed instead of the mode test mentioned above; i.e., instead of looking at whether or not the variable is bound, the value cell tag would be compared to the expected type, and if this succeeds, then the values could be compared.

Enhanced indexing scheme — The clause indexing scheme can be improved to test one or all arguments declared as inputs instead of, or, as well as, testing the first argument. This way, impossible unifications between the incoming arguments and the terms of the head goal of a clause will be determined at an earlier time, and thus less time is wasted in needless unifications. The scheme could be improved by eliminating unreachable instructions. For example, an argument which is ground when received (an input argument) will never execute the case for a **ref** variable in the the clause indexing code. It should be recognized that this is beneficial for input variables for which a type is not supplied and cannot be inferred by the code.

Quick indexing — Arguments defined as inputs which have declared types can have a significant effect on the determinacy of a clause (cref. § 2.4.2). If the argument was, for example, a constant, and only one clause unifies with this type, then that clause is the last alternative, and this is a necessary condition for a determinate clause. If an argument is declared to be an input, it must be bound, therefore, the number of strictly determinate clauses might increase. With declared modes and types, a procedure will often become completely determinate since only one clause is ever applicable according to the type of the input arguments. For instance, the **member/2** procedure of Figure 4.1 would be determinate if the second argument was defined as an input. In this example, the first clause is used if the second argument dereferences to the **nil.0** atom, and the second clause is used if the second argument is a **list.2** structure. For any other input, the procedure would fail immediately. Quick indexing would appear, like in WAM, at the beginning of the procedure as shown in Figure 5.1. Because all clauses are determinate, the following optimization techniques are now applicable: (1) eliminate the **switch** construct and alternative looping code, and (2) eliminate the state saving instructions.

```

if (arg2.type == string type)
    if (the value of arg 2 == the atom table ptr for nil.0)
        use clause 1;
    else
        return(fail);
else
    if (arg2.type == list_2 type)
        use clause 2;
    else
        return(fail);

```

Figure 5.1: Quick indexing scheme for `member(+,+)`.

Specialization of the `get` instructions — This optimization requires mode dependent `get` instructions to be written. For example, `get_const_read`, would be used for input arguments and, alternatively, `get_const_write` would be used for output arguments. The instruction classifier, `read`, implies that a comparison is to be made, and `write` corresponds to term construction. For input arguments, general unification is only required for the levels of a compound term for which the mode has not been specified; e.g., in the example above, although the second clause of `member/2` expects a list structure as the second argument, the terms of the list were not required to be bound.

Elimination of redundant testing — This optimization is an extension of the previous one. Since the type of an argument is tested during clause indexing, there is no sense in checking the type again in the specific `get` instruction. For example, if a list is expected and found, then the `get_list_read` instruction need not re-evaluate that the argument is a list, and it should not have to dereference the argument again. In fact, the procedure call could be eliminated completely if the mode and type tests had returned the location of the first term of the structure on the copy stack.

```
count([],0).
count([X|T],NUM) :- count(T,SUB), NUM is SUB + 1.
```

Figure 5.2: `count/2` predicate as it is typically written.

Eliminate trailing for functions — If all of the arguments can be declared to be strictly inputs or outputs, and the side effects of the call are tolerable, then the procedure becomes equivalent to a 'C' function. In this situation the procedure will always return determinately, but the instantiations it performed need not be trailed. Essentially, the goal is expected to perform an extra-logical evaluation such as I/O. This will be re-visited when the approach to writing 'C' functions to be used in a Prolog environment is discussed.

5.1.3 Optimization Of Determinate Procedures

Chapters 2 and 3 discussed the benefits of determinate procedures in a structure copying implementation (the call stack environments can be reclaimed immediately on completion of a determinate goal proof). There are several other optimizations which can be made to the PICG output when procedures are classified as determinate. This section discusses ways of finding determinate clauses and procedures, and lists the optimizations which can be made from this information.

A clause is defined to be determinate if it is the only alternative for proving the current goal, and it has only one correct solution. For example, the last clause of most procedures is determinate because when it is invoked, it is the only alternative left in the proof of this goal (c.f. § 3.3.2). A procedure is determinate if it will only ever produce one solution to a goal. For example, a predicate which counts the number of elements in a list is given in Figure 5.2. Although the first clause may not appear to be determinate, if the procedure is always expected to be called with a complete list as the first argument (i.e. the mode definition is `count(+,?)`) then the predicate is determinate because it will only ever produce one solution for a given list. Mode declarations are generally very helpful when looking for determinacy.

Without mode assignments, the basic rule for classifying a procedure as deter-

```

count([],0) :- !.
count([X|T],NUM) :- count(T,SUB), NUM is SUB + 1.

```

Figure 5.3: A strictly determinate version of the `count/2` predicate.

minate is defined by [18], and it states that a predicate is determinate if:

- each clause contains a cut operator (the last clause may be excluded from this test);
- each goal called after the cut is also determinate (each clause has only one solution).

The `count/2` predicate could be rewritten to include a cut in the first clause, and in this case the predicate is determinate regardless of the expected input modes. This version of `count/2` is shown in Figure 5.3. The PICG would generate a more complex function for the latter definition because the cut would have to be managed, however, both versions could be tuned to the same function as will be shown later. Mellish also provides an explanation for classifying other, more difficult to spot, procedures as determinate [18]. His scheme uses the classifications which are found easily through visual inspection, (for both determinate and non-determinate procedures), and an iterative propagation of this information through procedures which call these goals, to draw conclusions about their own determinacy. Essentially, each goal is defined, in equation form as:

1. **true** if they are known to be determinate,
2. **false** if they are known to be non-determinate, or
3. as a conjunction of the **true/false** determinacy of the subgoals. Unknown determinacies are represented by the subgoal predicates until the iteration produces a **true/false** value.

Each predicate is assumed to be determinate at the start of the deduction process. These values are substituted into the equations to produce new values. The substi-

tution is repeated iteratively until the classification settles. Examples are given in [18].

Once a procedure has been classified as determinate, the following optimizations can be performed:

Elimination of unreachable code — This optimization was listed already under the mode section as *quick indexing*, however, here, the focus is on removing unreachable code. If all clauses of a procedure are determinate, then each can return to the caller itself once it has been selected and proven, therefore, there is no need for the looping instructions which provide access to all clauses.

Elimination of calls to `manage_action` for determinate subgoals — If a clause is determinate because each subgoal called is determinate, then it can be re-written to call the subgoals itself, and to control its own execution. Therefore a clause:

a :- b,c,d.

could be written as:

... *call b; call c; call d* ...

This approach eliminates the need to make extra procedure calls through `manage_action`, and allows the parent to re-use argument lists which would not have to be established until the particular goal call is to be made.

This optimization can be generalized to the set of subgoals which are to be proven first, and which are all determinate; i.e., if the first subgoal(s) of a clause is determinate, then it can be called directly from the parent goal.

Determinate predicates as functions — A determinate procedure which is always called directly from its parent, (as done in the case described above), can be re-written as a 'C' function. These procedures are not required to have the standard PIC calling sequence because they are acting as imperative language functions, therefore, they can return output variables as the value of the function. If the bindings which result from this goal proof can be left after the call returns, then trailing can be ignored.

```

:
assign 0 to a temporary counter
while (continuu == true) do
  dereference argument 1
  if (argument 1 is the nil_0 atom) then
    continuu = false;
  else if (argument 1 is a list) then
    ++the value of the temporary counter
    adjust argument 1 to reference the tail of itself
  endif
endif
endwhile
dereference argument 2
if (argument 2 is bound) then
  if (the value of argument 2 == temporary counter) then
    return(determinate);
  else return(fail);
else assign <int,value of temporary counter> to argument 2
  return(determinate);

```

Figure 5.4: `count/2` determinate procedure in optimized 'C' form.

Optimization of non-logical goal calls — If a procedure is determinate, then built-in goal calls can often be placed in line. Earlier, a determinate procedure was said to call only determinate subgoals, and can be re-written to perform each call itself. However, built-in predicate calls, for example arithmetic evaluations often can be expanded directly in-line of the procedure. As an example, the `count/2` predicate given in Figure 5.3, can be optimized to the 'C' code given in Figure 5.4.

Finding clauses open to TRO — The new schemes for discovering determinancy

will open more procedures to tail recursion optimization.

Iterative procedures — If the last clause of a two clause procedure can be optimized with TRO, then the procedure can be turned into a **while** loop which tries to prove the first clause and the continuation goals. If this succeeds then the procedure returns, however if it fails then the initial goals of the second clause can be called before the program loops around. An example of this update was performed during the testing phase and is given in § E.2.1.

5.2 ‘C’ Functions in a Prolog Environment

For this thesis, a ‘C’ function in a Prolog environment will be a determinate, zero-arity predicate to be called by the PIC procedures which is written by hand in a more efficient format than the translator could produce, or because it is to perform an extra-logical task such as I/O. These predicates must have the standard calling sequence of a PIC procedure because they are most likely to be called through a closure in a continuation list. However, if the predicate is always being invoked directly from its parent, then the programmer can define the calling sequence himself. This section will look at an example taken from predicates already written for the built-in predicate library to show how to add new ‘C’ functions to a Prolog environment.

The example being considered is shown in Figure 5.5. The predicate `nl/0` is determinate, (it has only one solution which is to print a new-line character), and it is not passed any arguments. The variable `global_outfile` is a global pointer, defined in the target system, which defines the current output stream. The default stream is the ‘C’ `stdout` (standard output). This pointer is assigned a value when `set_output/0` is called assuming that `set_channel/2` has been previously called to assign a name to the output stream. In this example, the standard calling sequence of a PIC function is maintained, however, there is no resemblance between the body of this example and the general PIC format of Figure 3.3. The defined layout is not needed because there is no chance to fail on unification and there is only one possible solution. This example is typical of the operations which fall into this programming

```

action nl0(arg_list,continuation,global_stack,
           trail_stack,choice_point)
arg_list_type    arg_list;
contin            continuation;
g_stack          global_stack;
t_stack          trail_stack;
int              choice_point;
{
    if (global_outfile == NIL)
        fprintf(stdout, "\n");
    else
        fprintf(global_outfile, "\n");
    return(determinate);
}

```

Figure 5.5: PIC version of nl/0 built-in predicate.

category. The other built-in predicates which are of this type are:

1. `set_input/0` which simply opens `global_infile_name`, which is defined by `set_channel/2`, for reading,
2. `set_output/0` which simply opens `global_outfile_name`, which is defined by `set_channel/2`, for writing,
3. `close_input/0` which simply closes `global_infile`,
4. `close_output/0` which simply closes `global_outfile`, and
5. `cut_input/0` which assigns a global boolean variable, `cut_input`, to be true. This variable is tested by `read_char/1` when backtracking occurs.

5.3 Hand-compiling PIC

Hand-compiled PIC adds one or two features to what was classified as 'C' functions in a Prolog environment. The first addition is variable access. Some examples of the built-in predicates of this type are:

1. `set_channel/2`,
2. `nl/1`,
3. `read/1`,
4. `write/1`,
5. `add_statement/1`,
6. `del_statement/1`, and
7. *all of the arithmetic and testing predicates.*

The second enhancement allows backtracking; i.e., predicates with more than one alternative like:

1. `read_char/1`,
2. `add_statement_b/1`,
3. `del_statement_b/1`,
4. `spy_dyn/1`, and
5. `repeat/0`.

This section will use some of these examples to explain the process of writing hand compiled PIC.

5.3.1 Variable Access

In the previous section, `n1/0` was examined as a strictly ‘C’ function called in a Prolog environment. The first example of hand-compiled PIC is `n1/1` which prints as many new-line characters as defined by the first argument. The difference between these two functions is that the latter must read the value of the argument passed in the call. The mode declaration for this predicate is `n1(+)`, and it is known that this argument must be an integer. The first lines of the code for `n1/1`, shown in Figure 5.6, will determine if the mode and type of the argument are appropriate. This function can be simplified if the mode and type are assumed. In that case, the procedure will always return **determinate**. There are several lines in this example which must be explained before the user would be expected to write his own hand-compiled PIC version of a similar task.

1. All value cells (mini stacks of `obj_ptr`) are referenced by a pointer of type `obj_ptr*` (line 2).
2. In line 5, `unbound_integer(*deref)` is a macro which expands to:

```
(*deref == (obj_ptr)atom_arr[(int)integer] &&  
*(deref + value_offset) != NIL)
```

where:

```

action nl1(arg_list, continuation, global_stack,
           trail_stack, choice_point)
arg_list_type    arg_list;
contin           continuation;
g_stack          global_stack;
t_stack          trail_stack;
int              choice_point;
{
    int           index, num;                (1)
    obj_ptr       *deref;                    (2)
    boolean       bound;                     (3)
    /* get the dereferenced location of the first argument. */
    deref = dereference(&arg_list[0], &bound); (4)
    /* if (the first argument is a ground integer) then continue */
    if unbound_integer(*deref) {             (5)
        num = *(int *)*(deref + value_offset); (6)
        if (global_outfile == NIL)           (7)
            for (index = 0; index < num; index++) (8)
                fprintf(stdout, "\n");         (9)
        else                                  (10)
            for (index = 0; index < num; index++) (11)
                fprintf(global_outfile, "\n"); (12)
        return(determinate);                 (13)
    }                                         (14)
    else return(fail);                       (15)
}                                             (16)

```

Figure 5.6: Optimized PIC version of the nl/1 predicate.

- (a) `atom_arr` is an array which contains pointers into the atom table. The array elements are assigned to correspond to a set of enumerator types of which `integer` is one. This index must be type cast to an `int` to compile.
 - (b) `( *deref == (obj_ptr)atom_arr[(int)integer] )`, tests to see if the tag associated with the dereferenced value cell is the `integer` tag.
 - (c) `( *(deref + value_offset) != NIL )`, tests the value pointer to ensure that the argument is ground; i.e., `value_offset` moves the value cell pointer from the tag field to the value pointer field.
3. In line 6, `num = *(int *)*(deref + value_offset)`, is type casting the value pointer of the dereferenced cell to an integer pointer, and assigning the destination of that pointer to the local variable `num` (line 1).

In this example both modes and types are being considered to optimize the procedure.

The second example will examine the code for `=/2`. It shows how the available unification functions can be used in hand-compiled code. The predicate can be represented in Prolog as:

```
unify(X,X).
```

This would translate to the following WAM instructions:

```
get_value(argument 2, argument 1)
proceed
```

The optimal form replaces the call to `get_value` with the code expansion of that function as shown in Figure 5.7.

5.3.2 Backtrackable Hand-compiled PIC

Of the backtrackable hand-compiled PIC procedures listed at the beginning of the discussion, two contain tail recursive last clauses (`repeat/0` and `spy_dyn/1`), and the others all return determinately when processing their last clause (`read_char/1`,

```

action unify2(arg_list, continuation, global_stack,
              trail_stack, choice_point)
arg_list_type      arg_list;
contin              continuation;
g_stack             global_stack;
t_stack            trail_stack;
int                 choice_point;
{
  obj_ptr            *deref1, *deref2;
  boolean            bound1, bound2, status;
  deref1 = dereference(&arg_list[0], &bound1);
  deref2 = dereference(&arg_list[2], &bound2);
  status = unify(deref1, deref2, global_stack,
                 trail_stack, choice_point);
  if (status == true)
    return(success);
  else
    return(fail);
}

```

Figure 5.7: Hand-compiled version of the unify/2, =/2, predicate.

add_statement_b/1, and **del_statement_b/1**). In this section, one predicate from each of these categories will be examined as examples of optimized backtrackable hand-compiled PIC procedures.

The **repeat/0** procedure is defined in Prolog as:

```
repeat.  
repeat :- repeat.
```

In this predicate the first clause always succeeds. The second clause is tail recursive, however, because the purpose of the predicate is to repeatedly create choice points, the procedure is not determinate. The hand-compiled PIC code for this function is given in Figure 5.8. After the first clause succeeds, the received continuation is passed to **manage_action** and the top of the copy stack on entry into this procedure is sent to define the newest choice point. When **manage_action** returns, the result is used to determine if the second clause should be tried. In the case of a fail, the loop indicator **contin** is left unchanged, and the copy and trail stacks are reset because the execution has backtracked into a choice point. Otherwise, a **success** status is returned to the caller to indicate that the entire continuation passed to this procedure has been proven successfully. In this example, the standard calling sequence is preserved, but most of the function body has been changed as a result of the following optimization techniques:

1. *updates to local variable declarations,*
2. *elimination of unreachable instructions, and*
3. *recognition of the iterative, tail recursive nature of the procedure.*

The second example is **read_char(+)** which is defined in Prolog as follows:

```
read_char :- read the first character of the current input stream.  
read_char :- return the character read during the execution of the  
               first clause unless a call to cut_input/0 has been  
               made in the mean time.
```



```

action repeat0(arg_list,continuation,global_stack,
               trail_stack,choice_point)
arg_list.type   arg_list;
contin          continuation;
g_stack         global_stack;
t_stack        trail_stack;
int             choice_point;
{
  boolean       continu,determ;
  int           backtrack_level,global_top,trail_top,new_choice_pt;
  action        next_action;

  global_top = global_stack->top; /* important for clean up */
  trail_top = trail_stack->top;
  continu = true;
  new_choice_pt = global_stack->top;
  while (continu==true) {
    next_action = manage_action(continuation,global_stack,
                               trail_stack,new_choice_pt);
    if (next_action != fail) continu = false;
    else clean_up(global_stack,global_top,trail_stack,trail_top);
  }
  return(next_action);
}

```

Figure 5.8: Optimized version of the hand-compiled PIC function `repeat/0`.

```

action read_char1(arg_list, continuation, global_stack, trail_stack, choice_point)
: same argument declarations as in the other examples
{
    obj_ptr      *character;
    boolean      bound, determ, continu;
    char         c='\0';
    action       next_action;
    FILE         *infile;
    int          backtrack_level;

    if (global_infile == NIL) infile = stdin;
    else infile = global_infile;

    character = dereference(&arg_list[0], &bound);
    if (*(character+2) == (obj_ptr)atcm_arr[(int)file_0])
        if (feof( infile )) next_action = success;
        else next_action = fail;
    else
        if looking for an expression end then get and compare the next character to '.'
        else if looking for end-of-line then get and compare the next character to '\n'
        : assign next_action accordingly
        /* endif */

    if (next_action == success) {
        next_action = manage_action(continuation, global_stack,
                                   trail_stack, choice_pt);

        if (next_action == fail)
            if (c != '\0' && cut_input == false) ungetc(c, infile);
            else cut_input = false;
    }
    else if (c != '\0') ungetc(c, infile);
    return(next_action);
}

```

Figure 5.9: Optimized version of the hand-compiled `read_char/1` predicate.

The optimized hand-compiled PIC version is given in Figure 5.9. In this example, the ground first argument is dereferenced and compared to the type of the next character in the current input stream. If this match is successful, the received continuation is passed in a call to `manage_action`. A choice point is not created in this procedure because, although the procedure is non-determinate (the continuation has been proven), the second clause always returns immediately to the caller. If the program is backtracking, the execution state is reset by the nearest choice point.

If the first clause fails while comparing the input character with the type defined by the arguments, the second clause returns the character to the input stream, and a `fail` status is returned to the caller. If `manage_action` returns a `fail` status, backtracking is in progress, and the second clause proceeds according to the value of the global, boolean variable `cut_input`. If this parameter has been set to `true`, then the character read in the execution of the first clause, must not be returned to the input stream because a `cut_input/0` goal call has been made since this `read_char/1` goal was called. Otherwise, the character is returned before the `fail` status is relayed to the caller.

Although these may seem like simple examples, it is expected that the user would not be inclined to fully optimize, or hand-write predicates which are much more complex than these. In fact, that would defeat the purpose of offering the opportunity to code in a declarative language the portions of the system which are best suited to that programming style. However, of the supplied built-in predicates, `spy_dyn/1` is particularly useful for understanding unification and backtracking in hand-compiled PIC, and `write/1` or `is/2` show how variables are stored and decomposed for evaluation.

Chapter 6

PIC and PICG Analysis

The PICG runs in the PDSS environment of MProlog [8], on a SUN UNIX workstation, and produces 'C' code to run on the same system. A set of built-in PIC predicates is provided covering the following general areas: (1) control, (2) arithmetics, (3) testing, (4) dynamic clauses, and (5) I/O. These are listed and defined in Appendix D. The steps involved in generating PIC code are given in § E.1.

The PICG translator is written entirely in MProlog. The main concepts of the compiler, and the unification support software (written in 'C') were developed and running simple predicates, like `member/2`, in less than three weeks. It took another three weeks to add the built-in predicates to the system with most of the effort going to the extra-logical, arithmetic and I/O features. This first version only operated in the determinate mode (cref. § 3.3.2). The emphasis at that time was to develop a translator for a multi-language development environment where the Prolog code segments would be small and sequential in nature. The programmer would, therefore, be responsible to be cautious of the "one solution" criterion. Once this version was debugged, the extensions required for the less restrictive mode were easily implemented.

In order to evaluate the performance of a system by testing, the following factors must be considered: (1) the experiment itself, (2) the system type, (3) the system load, (4) the test cost, and (5) the measuring tools. Basically, the experiment should be typical and representative, and all tests should be run in the same system

environment. Four Prolog programs were used to test the PICG output.

Benchmark Test — The first test program is a naive reverse benchmark test which was supplied as part of a sample program set for the PC version of ALS Prolog [7].

Exam Scheduler — The next test program is a typical generate and test problem which exercises the backtracking control strategy and trailing, as well as dynamic facts.

An Interpreter — The third test program is a LISP interpreter. This test exercised the clause indexing scheme, the cut, and tail recursive predicates.

Large Prolog Project — The final test program is a complex document conformance analyzer developed for the ODA (Office Document Architecture) International Standard [10]. This program executes almost completely non-determinately, thus producing an immense search tree with many choice points. Its execution would put a strain on any Prolog system. This program was used to test complex structures, non-determinism, meta-evaluation, and dynamic facts.

All tests were run on a diskless SUN 3/60 workstation with 16Mbytes of memory. The tests were run in an unloaded UNIX environment. Errors introduced in timing measures (typically due to page swapping) were minimized by calculating an average value from several iterations.

6.1 Benchmark Test

The standard benchmark used to compute LIPS (logical inferences per second) for Prolog systems is the naive reverse predicate [7]. The program was taken directly from the examples set of the PC version of ALS Prolog and can be found in § E.2. The naive reverse procedure is used in `nrev(+,-)` mode for the test. The predicate takes a list as input and returns the reverse of the input list. The ALS literature indicates that the list length for the test should be at least thirty elements long. The accuracy is improved if the number of iterations of the reverse call is sufficient

	non-determ	determ except for/3	optimized	MProlog interpret	MProlog compiled
LIPS	3319	3299	3755	2730	2786
maximum memory use in Kbytes	14	13	13		
average total time to run the test program in seconds	15.2	15.2	13.4		

Table 6.1: Results of the benchmark test.

to make the program run for at least one second. The constant `cputime`, requested by the `time/1` predicate, was provided by the UNIX 'C' function `ftime`.

The translation was performed in both modes of operation, however, the determinate version of the `for/3` predicate had to be replaced with the non-determinate output of the same procedure. The assumptions made in the determinate translation are incorrect for this predicate¹. A 'C' mainline, and a CPI equivalent were added to the PIC program as shown in Figure E.2. The average results of the test runs are shown in Table 6.1.

The program produces a LIPS measure itself and the other results listed in the table were obtained with the UNIX `time` facility. MProlog does not support a `cputime` value. For these results, the `execute/1` predicate was used to provide the times taken to perform the `runtest/2` and `runcontrol/2` goals. LIPS were calculated from these times using the same equation defined by the naive reverse program. Any overhead introduced by the `execute/1` goal would be eliminated by taking the difference between the naive reverse time and the control time if the overhead is static. If the overheads are linear with execution time, then the value given for LIPS of the MProlog version would be low.

The results show that all PIC versions are more efficient than MProlog regardless of whether the program is interpreted or compiled. The non-determinate case was

¹The predicate is called within a failure driven loop (cref. § 3.3.2).

slightly faster than the determinate case because less function returns would have been performed during the naive reverse portion of the program. The overall times of both these programs were approximately the same. The determinate case did show a reasonable savings in memory requirements by using one Kilobyte less in its maximum real memory allocation [9]. Several predicates return immediately in this version, and hence their call stack space is recovered. For example, the **genlst/2**, **append/3**, and **nrev/2** procedures all return determinately from both clause alternatives.

The table shows one other entry which represents a hand-optimized case of the determinate program. Only the **nrev/2** and **append/3** procedures come into the performance measure since the control time is subtracted in the LIPS calculation. One of the optimization techniques involved two clause determinate procedures for which one clause is tail recursive (cref. § 5.1.3). In this program, **append/3** is being used determinately, therefore, it fits this description, and was optimized accordingly. The before and after 'C' code is given in § E.2.1 to show the relative simplicity of the update. It took less than ten minutes to have the new version running, and the improvement in efficiency is significant (approximately 14%).

Another optimization was added to the control code. The subgoal **I1 is I + 1** was moved in line of the **for/3** PIC function. This update is shown in § E.2.2. To see any benefit, the number of iterations had to be increased significantly. At 1000 iterations, the time taken to run the control loop in the determinate version was approximately 1.6 times longer than the control loop in the optimized version. For 750 iterations this factor dropped to 1.3.

6.2 Exam Scheduler

The **generate-and-test** program is listed in § E.3. The program schedules PHD oral examinations according to the committee members and their available time slots by selecting a time which is convenient to all members, and which does not place one professor in two exams at the same time. The program decides which time slots are common among a exam committee, and marks these as potential

Test	non-determinate	determinate	MProlog
1	1.33	1.33	2.08
2	1.88	1.85	2.92
3	4.16	—	6.01
4	6.34	—	8.77

Table 6.2: Execution times (sec) for the Exam Scheduler.

Test	non-determinate	determinate
1	13Kbytes	11Kbytes
2	12Kbytes	12Kbytes
3	13Kbytes	—
4	15Kbytes	—

Table 6.3: Maximum memory requirements for the Exam Scheduler.

times for the corresponding exam. It then tries to allocate particular times for the exams by selecting one of the potential slots, and testing that there are no conflicts between the committee members because of exams which are already scheduled. If a problem exists, then the program backtracks as far as necessary to select a new time for a committee and repeats the test. The program establishes several choice points during the scheduling process, and is therefore a good general test of the control scheme. This program tests the continuation passing mechanism, backtracking (trailing and cleaning up as a result of returning to a choice point), and dynamic fact assertions and retractions which are used extensively throughout the program.

The non-determinate PIC output successfully executed the program for all four test sets; i.e., the program produced the same solutions, and followed the same search tree path as the interpreted program. The execution times of these two versions are shown in Table 6.2. The first and fourth test sets introduce data for which a conflict free schedule can be obtained. The second test set fails early in the program when potential time slots are being determined because there is one

committee for which there is no common available time. The third test set fails to schedule the committees, but must perform an exhaustive search of all potential time slots before it fails. A comparison of the timing for these two versions shows that the PIC version consistently takes at least 30% less time to execute. This gain in execution efficiency is much higher than was expected since the effort went more into keeping the output simple and consistent than towards making the programs run efficiently.

The determinate PIC version of the program was only able to correctly execute the first and second test sets because several clauses are incorrectly classified as determinate. This version used two Kbytes less memory space than the non-determinate program to prove the first test set as shown in Table 6.3, but required the same amount of space for the second test. The latter test fails before the actual scheduling occurs, therefore the program does not get a chance to execute the non-determinate part of the program.

There are not very many opportunities for manual optimization in this program. For example, the `member/2` predicate is not being used determinately, therefore, it cannot be optimized the way that `append/3` was in the previous test. The `all_avail_in_slot/2` predicate can be optimized in this way, and because each subgoal is being used determinately, these calls can be made directly from this function. This optimization was performed and is given in § E.3.1, however, because the processing performed here does not represent a significant part of the overall execution, no improvement in efficiency was noticed (on average, it was at most one tenth of a second faster for the four test sets).

6.3 Interpreter Test

The LISP interpreter test, which is listed in § E.4, was used to test and debug the clause indexing scheme. It is an interactive program which, therefore, does not provide any interesting timing data, but its determinate nature allows for several optimizations to be made. In particular, the predicate `built_in/1` is very simple and is always used in `built_in(+)` mode where the argument is always a string

or quoted term. This predicate can be simplified to what is shown in Figure E.1. There are several determinate tail recursive predicates which can be turned into simple loops: (1) **reverse/3**, (2) **last/2**, (3) **append/3**, and (4) **print_tail/1**. The clause indexing scheme of **solve/3** can be simplified because each clause except the last is looking for a **list_2** structure, or enhanced by indexing on the first element of this list which is a unique atom for each case. Replacing **built_in/1** and simplifying **solve/3** alone reduced the size of the ASCII source program by 19Kbytes, however, no change was found in the size of the object file which includes all of the support software as well as the PICG produced program.

6.4 Large Prolog Project

The final test program provided an extensive overall exercise for the PICG and its output. The program itself is too large to append to this thesis, but it required several enhancements and additions to be made to support the program, and caused many problems to surface which were hence corrected. The coverage of this last test program is defined by the lists which follow.

Enhancements to the PIC Development Environment

1. The PICG was updated to read operator declarations, and thereafter correctly parse such terms or goals. The operators had to be added to the set of atom table entries to support the corresponding compound term construction.
2. The PICG was extended to accept *value* declarations. A value in MProlog is analogous to a global variable in imperative programming because a ground term representation is assigned (not unified) to a value. A value is examined using the **get_value(name, rep)** predicate where *rep* is unified with the representation of the value. A value is assigned with the **set_value(name, rep)** goal call.
3. Meta-evaluation was added and used via the **evaluate/1** predicate (cref. § 3.3.5). This also increased the size of the atom table and its entries. All

predicates have to be stored in the atom table along with their function pointers so that they can be called correctly from a goal structure built on the copy stack.

4. Several optimizations were added to the fixed code portions of each PIC function to reduce the size of the output, and to make the translation more flexible. One update which was not implemented, but would be a worthwhile extension, would be to predetermine the size of each argument list. At times, this test program creates compound terms which require hundreds of temporary locations since each sub-structure of a compound term is unified with one temporary location (as an example, one term needs 601 temporary locations for its construction). Currently, the size of all argument lists is fixed, but the PICG could be extended to perform a multiple pass translation which would allow the size of each argument list to be determined. The user would no longer have to check for out of bound references, and update the code manually with this revised PICG.

Observations

1. Several features of the PIC design were verified: (1) nested structures, (2) nested disjunctions, (3) temporary variable allocations, (4) choice point assignments, and (5) clause indexing.
2. Tail recursion optimization was tested for cases where the recursive clause was not necessarily the last alternative, and therefore the trailing and backtracking had to be managed by the same procedure.
3. One function in the test program was deemed untranslatable. However, from the output which was generated, it was easy to manually adjust the code to act as desired. This function is shown in Figure 6.1. This predicate is used to determine a procedure which corresponds to an operator found as part of a compound structure which itself is being evaluated by meta-evaluation. For example, the $\hat{}$ operator implies that the `hat/2` predicate should be invoked by the meta-evaluation. The PICG and executing PIC environment cannot relate

```

function_lookup(^,hat) :- !.
function_lookup(",",arglist) :- !.
function_lookup(".",dot) :- !.
function_lookup(' ',spotset_select) :- !.
function_lookup("//,concat) :- !.
function_lookup(length,length2) :- !.
function_lookup(F,F) :- !.

```

Figure 6.1: `function_lookup/2` predicate.

the `hat/0` atom with the `hat/2` procedure as is expected by the programmer. Each special case listed in the predicate (all but the last alternative), and any function whose arity is not the same as the predicate to which the second argument of this goal is bound, had to be manually updated to reflect the correct arity of the desired function. For example, to unify the second argument (the associated predicate representation) of the first clause, the translator would generate the line:

```
GET_STR( hat_0, &arg_list[2*1], ... );
```

which must be manually changed to:

```
GET_STR( hat_2, &arg_list[2*1], ... );
```

Each of these atoms represents a different entry in the atom table which is referenced by the corresponding identifier (either `hat_0` or `hat_2`).

The test program reads a collection of facts which describe a document, and determines if this description conforms to the standard. Three test files were used to measure the performance of the PIC program. Each set of data represents a conforming document, and the tests become increasingly more complex from one test to the next. The first test set describes a document with only 9 facts, the second test uses 16 facts, and the third test document has 24 facts in its description.

Test	Test Size (# facts)	PIC	MProlog
1	9	82.4	217.57
2	16	191.3	635.22
3	24	352.5	1127.25

Table 6.4: Execution times (sec) for the ODA Analyzer.

The timing measures of these tests are given in Table 6.4. The PIC program ran approximately three times faster than the MProlog interpreter in all cases. This improvement in efficiency is most likely due to the fact that terms on the read stack are known to be ground, and that the functions which perform dynamic assertions use this knowledge to minimize the amount of copying required to store clause prototypes. Unfortunately, the PIC program would run out of free memory when executing larger examples, while MProlog was able to complete the proof of the same example. A fourth test file was used to verify the correctness of the program. This file was similar to the test with 16 facts, but was altered so that it would not conform to the standards. The PIC version was able to recognize that the file was inconsistent and failed appropriately. A timing measure showed that this test ran with the same (approximately 3 times) improvement over the MProlog interpretation of the same test file.

Chapter 7

Conclusions

The development of *Prolog in 'C'* was triggered by arguments in support of dual logic and imperative programming environments. Here, the emphasis was placed on producing a common language target system. This was done to enhance the portability of the system as well the simplicity of integrating the two programming techniques. For implementation, this corresponded to the definition of a conventional control structure for Prolog and the translation of logic programs to sets of 'C' functions. Two major goals were defined for the PIC code:

1. each PIC procedure would be invoked in a consistent manner thus simplifying the interface of PIC and hand-written PIC functions, and
2. each PIC procedure would be produced in a straightforward and typical, imperative programming style to make the program: (1) more readable, (2) more understandable, and (3) open to manual tuning.

The work focused on the development of the PICG (*Prolog in 'C' Generator*), and the PIC execution environment. The expected end user was described as someone who is working on a 'C' system with modules well suited to declarative programming, or someone wanting a method to obtain a first-step, workable, 'C' implementation of a Prolog program. A control strategy based on continuation passing, and a unification procedure modeled after WAM compilation were combined to define the PIC execution which, in turn, set the requirements for the translator. The PICG

and the support PIC software were developed, and their capabilities were studied and discussed in Chapter 6. Four test programs were run for the purpose of:

1. evaluating the correctness of the PIC code execution,
2. evaluating the coverage of the translator and the system as a whole; i.e., the range of declarative programs which could be made to execute in a PIC environment,
3. measuring the performance of the target programs, and
4. measuring the costs (programmer time spent) and savings (performance improvements) of manual optimizations.

The results of these tests indicate that the PIC programs run correctly in all instances when they are translated with no assumptions on clause determinacy. The programs output the expected results, and would follow the same execution path as the MProlog interpretation of the same program. However, with respect to coverage, the input Prolog programs are currently restricted as follows (the limitations have been listed in the expected order of decreasing complexity of their eventual implementation):

1. a `!/0` goal must not appear within an `or/2` structure (cref. § 3.3.3);
2. the PIC execution environment does not have the ability to recognize and correlate atoms with the same functor but different arity;
3. only one `!/0` goal may appear in a given clause;
4. dynamic rule assertions and retractions were never implemented;
5. the reader accepts only ground terms;
6. 'C' has trouble with special characters, therefore the user must update the translator to accept special character operators, and output another, consistent representation for the symbol:

7. the built-in predicate set is smaller than average for Prolog systems especially with regards to I/O.

The performance measures of the tests showed that PIC executes reasonably in comparison with the MProlog interpreter. The execution times were slightly but consistently faster for the PIC versions. However, the memory demands were higher and MProlog is not known for being particularly memory efficient [3]. The performance can be improved with manual optimizations (cref. § 5.1), as shown during the testing phase. Memory savings and garbage collection did not weigh heavily on the implementation of PIC since the emphasis was placed on allowing the programming mode to switch easily from 'C' to Prolog and back again as opposed to having large Prolog programs execute as 'C' code. When a Prolog environment returns to the 'C' code through a CPI function, all of the corresponding information structures are deallocated (cref. § 3.1).

Manual optimizations were performed on a few specific predicates. The examples represent common declarative programming styles and were all easily recognizable from basic knowledge of the program execution and the suggested descriptions of § 5.1. The tuning involved very little effort because, for the most part, the variable classifications and unification functions were left as is, and only the control was changed. The control forms the most straightforward part of a PIC function as far as a sequential language programmer is concerned. The update of the `built_in/1` predicate (cref. § 6.3), on the other hand, defines a strict replacement of the translator output with a simple and efficient hand-written PIC function. Again, this replacement required little effort for an experienced PIC programmer.

Some optimizations could be performed automatically by the translator with the introduction of mode declarations, and determinate clause or procedure classifiers. The requirement to keep a standard format for every PIC function would have to be eliminated, and several changes and enhancements would have to be made to the translator code; for example, the translator would have to recognize a mode declaration statement, and act on it appropriately. This represents several modifications in the translator [20, 14, 32]. The automatic optimizations of PIC can also be improved by implementing a multiple pass translation. This is also a costly exercise

for both time and memory requirements of the transition process.

The goals set forth in this thesis represent the definition of a workable, and practical multi-language programming environment which combines both the declarative programming of Prolog and the sequential programming of 'C'. The development of the PIC translator and support software for a running PIC program has been completed successfully to prove that the software generations can be combined to produce a usable and potentially powerful approach to programming.

References

- [1] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MASS:MIT Press, 1985.
- [2] I. Bratko, *PROLOG Programming for Artificial Intelligence*. Reading, MASS:Addison-Wesley, 1986.
- [3] L. Burkholder, R. W. Gillingham, M. Kato, J. Gordon, R. Gustafson, D.G. Shafer, and C.K. Townsend, "14-Product Wrap-Up: PROLOG for the People," *AI EXPERT*, Vol. 2, No. 6, pp. 63-84, June 1987.
- [4] R.W. Coffin and J.W. Roach, *An Implementation of D.H. Warren's PROLOG Machine on a Vax 11/780*, TR VT-86-17, Dept. of Computer Science, Virginia Tech, Blacksburg, VA, August 1986.
- [5] Arity Corporation, *The Arity/Prolog Language Reference Manual*. Concord, Massachusetts:Arity Corporation, 1988.
- [6] Arity Corporation, *Using the Arity/Prolog Interpreter and Compiler*. Concord, MASS:Arity Corporation, 1987.
- [7] Applied Logic Systems INC, *ALS Prolog Technical Reference Manual*. Syracuse, NY:Applied Logic Systems INC, 1988.
- [8] Logicware Inc., *MProlog Language Reference, Release 2.1*. Toronto, CAN:Logicware Inc., September, 1985.
- [9] Sun Microsystems INC, *Programming Utilities for the Sun Workstation*. Mountain View, CA:Sun Microsystems INC, Feb. 1986.

- [10] G.M. Karam, *The FODA Conformance Analyzer: Prototyping with Prolog*, TR SCE-88-12, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, CANADA, August 1988.
- [11] G.M. Karam, "An Icon-Based Design Method for Prolog," *IEEE Software*, Vol. 5, No. 4, pp. 51-65, July 1988.
- [12] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*. Englewood Cliffs, NJ:Prentice-Hall, 1978.
- [13] F. Kluzniak and S. Szpakowicz, *Prolog for Programmers*. London:Academic Press, 1985.
- [14] H. Komatsu, N. Tamura, Y. Asakawa, and T. Kurokawa, "An Optimizing Prolog Compiler," In *Logic Programming '86 - Proceedings of the Fifth International Conference on Logic Programming. Lecture Notes in Computer Science #264*, New York, NY:Springer-Verlag, pp. 104-115, June 1986.
- [15] J. Laventhol, *Programming in POP-11*. Boston:Blackwell Scientific Publications, 1987.
- [16] D. Maier and D.S. Warren, *Computing with Logic — Logic Programming with Prolog*. Menlo Park, CA:Benjamin/Cummings, 1988.
- [17] C. Mellish and S. Hardy, "Integrating Prolog in the POPLOG Environment," In J.A. Campbell, editor, *Implementations of Prolog*. New York, NY:Ellis Horwood Limited, pp. 147-162, 1983.
- [18] C.S. Mellish, "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming*, Vol. 2, No. 1, pp. 43-66, April 1985.
- [19] C. Muller, "Modula-Prolog: A Software Development Tool," *IEEE Software*, Vol. 3, No. 6, pp. 39-45, Nov. 1986.
- [20] J.F. Nilsson, "On the Compilation of a Domain-Based Prolog," *Information Processing*, 1983.

- [21] N. Nilsson, *Principles of Artificial Intelligence*. Los Altos, CA:Morgan Kaufmann, 1980.
- [22] L. Odette, "How to Compile Prolog," *AI Expert*, Vol. 2, No. 8, pp. 48-57, August 1987.
- [23] F. G. Pagan, "Converting Interpreters into Compilers," *Software Practice and Experience*, Vol. 18, No. 6, pp. 509-527, June 1988.
- [24] G. Pitette, "AdLog: an Ada components set to add logic to Ada," In *Proceedings of the Seventh Joint Ada Europe/ Ada TEC Conference*, pp. 174-188, 1988.
- [25] E. Sandewall, "Conversion of Predicate-Calculus Axioms to Corresponding Deterministic Programs," *IEEE Transactions on Computers*, Vol. C-25, No. 4, pp. 342-346, April 1976.
- [26] L. Sterling and E. Shapiro, *The Art of PROLOG — Advanced Programming Techniques*. Cambridge, MASS:MIT Press, 1986.
- [27] Complete Logic Systems, *Trilogy User's Manual*. North Vancouver, BC:Complete Logic Systems, 1987.
- [28] E. Tick, *An Overlapped Prolog Processor*, TR SRI-83-308, SRI International, Menlo Park, CA, October 1983.
- [29] M. Wand, "Continuation-based Program Transformation Strategies," *Journal of the ACM*, Vol. 27, No. 1, pp. 164-180, January 1980.
- [30] D.H.D. Warren, *An Abstract Prolog Instruction Set*, TR SRI-83-309, SRI International, Menlo Park, CA, October 1983.
- [31] D.H.D. Warren, *Applied Logic-Its Use and Implementation as a Programming Tool*, TR SRI-83-290, SRI International, Menlo Park, CA, June 1983.
- [32] J.L. Weiner and S. Ramakrishnan, "A Piggy-back Compiler For Prolog," In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 288-296, Atlanta, Georgia, June 1988.

Appendix A

Supplementary Background Information

A.1 Interpretation of an Example Query

A query proof uses the program facts and rules; and, if successful, determines a value for each variable in the original goal. The root node of the graph analogy was a representation of the query goal. In the stack implementation of Figure 2.3, the first goal placed on the *resolution stack* is the query goal. This stack contains all goals which are yet to be proven by the execution of the interpreter. The following discussion provides a step-by-step description of the query `child(cathy,A)` in terms of the pseudo-code given in Figure 2.3.

1. Before starting the WHILE loop of the interpreter, see Figure 2.3, an *instance* of the query goal is pushed onto the *resolution stack*. This stack contains the goals which are yet to be proven by the interpreter. The *instance* of the goal is a description from which the predicate and arguments can be determined.
2. On entering the WHILE loop, the goal at the top of the resolution stack is the query goal. This is the *current goal* of the proof.
3. The next step is to search the program for *clauses* with the same predicate as the *current goal*; `child/2`. This is analogous to making a “call” to the

`child/2` procedure, and passing to it the arguments: (1) `cathy`, and (2) `A` which is an unbound variable (one which has not yet been assigned a value). There are two rules in the program, rule #1 and #2, which match the query goal.

4. When more than one branch from a node exists, (i.e. the matching procedure contains a collection of more than one clause), the execution has reached a *choice point*. Prolog tries to solve the query by taking the paths defined by the clauses in a top down manner. Rule #1 is used first; but, before the proof continues, the interpreter recognizes rule #2 as an alternative path to the solution.
5. The arguments passed in the call to `child/2` are *unified* (matched) with the terms of rule #1. These terms are analogous to the local variables of a Pascal procedure; that is, the variables are created on the *execution stack* much like the Pascal variables are allocated in the procedure's activation record on the Pascal call stack. Unification produces the most general substitution of `X=cathy` and `Y=A`. The local variables of the clause are *instantiated* (updated) to reflect this substitution.
6. The rule introduces the subgoal `mother(A,cathy)` which reflects the substitution required to unify the rule head goal with the current goal. This goal instance is pushed onto the resolution stack, and the execution loops back to the top of the WHILE loop.
7. `mother(A,cathy)` is now the goal which is currently at the top of the resolution stack, and is therefore the *current goal* of this loop.
8. There is only one clause, fact #2, which matches this goal; i.e it has a `mother/2` predicate. However, it is clear that `mother(A,cathy)` will not unify with `mother(betty, john)`. The interpreter has failed to prove the query with this path, and must now *backtrack*. In effect the `mother/2` procedure was "called", but it failed to match the arguments passed to it with any of its clauses.

9. The interpreter recalls that a choice point was marked at the root node. The execution state is reset to correspond with the beginning of this goal's proof: the execution *backtracks* to the previous goal which is, in this case, the query. The first alternative path is tried, which implies that rule #2 is tried. This is analogous to having the *mother/2* procedure return a failure status to the calling procedure, *child/2*. Therefore *child/2* tries to unify the arguments passed to it with the next alternative clause which is rule #2.
10. Rule #2 introduces the *subgoals* *sibling(cathy, B)* and *child(B, A)* which are pushed onto the resolution stack such that the left-most goal is at the top of the stack.
11. The proof continues with the consideration of the goal *sibling(cathy, B)*. This goal might match with fact #1 or rule #3, therefore this node represents a choice point. Fact #1 is tried first, and matches the goal if *B=john* as shown in Figure 2.2. The *B* variable is local to rule #2, and therefore this instantiation also affects the value of *B* passed to the second subgoal, *child(B, A)*; i.e. unification replaces all occurrences of *B* on the resolution stack leaving *child(john, A)* at the top.
12. The goal *child(john, A)* also has two potential paths, one created by rule #1 and another by rule #2. The proof has reached another choice point which is defined to be *nearer* than the one marked in the previous step. This means that when backtracking, this node is explored before the other is considered. In the Pascal analogy, the failure is returned to this procedure before any other. In this example, however, neither choice point will be considered because a solution will be found without backtracking to these nodes.
13. The next step is to unify fact #2 with the current goal, *mother(A, john)* which replaces *A* with *betty*.
14. The resolution stack is now empty, therefore the execution stops, and the interpreter indicates that the query can be proven when *A* is instantiated to *betty*.

A.2 Data Representation Examples

This section illustrates how terms are constructed and referenced using two common techniques, structure sharing and structure copying, to data representation. The example, adapted from one in [4], uses the following code segment:

$u \text{ :- } a(X), b(Y), c(X, Y), d(X), e(Y).$ (1)

$a([]).$ (2)

$b([]).$ (3)

$c(W, W).$ (4)

$d([a]).$ (5)

$e([b]).$ (6)

A.2.1 Structure Sharing

The example program above will be used to illustrate the construction and instantiation of variables in a structure sharing implementation. There are four compound terms, structures, in the program which are illustrated in Figures A.1, A.2, A.3, A.4. Execution will effect the terms as follows:

1. Procedure $u/0$ calls procedure $a/1$ with the argument $X: \langle \text{undef}, \text{undef} \rangle$.
2. Procedure $a/1$ unifies the incoming unbound variable successfully with a list structure containing a single variable. As a result of unification the $X: \langle \text{undef}, \text{undef} \rangle$ argument becomes $X: \langle \text{pointer to prototype \#1, pointer to a new value cell on the structure stack initialized to 0: } \langle \text{undef}, \text{undef} \rangle \rangle$. The value cell has been labeled 0 to indicate its location on the stack.
3. Procedure $u/0$ calls procedure $b/1$ with the argument $Y: \langle \text{undef}, \text{undef} \rangle$.

Figure A.1: Prototype #1 developed for equation (2)

Figure A.2: Prototype #2 developed for equation (3)

Figure A.3: Prototype #3 developed for equation (5)

Figure A.4: Prototype #4 developed for equation (6)

4. Procedure **b/1** instantiates **Y** to **Y: <pointer to prototype #2, pointer to a new value cell on the structure stack initialized to 1: <undef, undef>>**.
5. Procedure **u/0** calls procedure **c/2** passing it **X** and **Y** which now represent list structures.
6. Procedure **c/2** is true if the two incoming arguments can be unified. Using the steps given in Section 2.1.3, the functor/arity of the two structures are matched, and since both are **list/2**, the first arguments are unified. This causes the value cell at location 0 to become **0: <ref, pointer to location 1>**, and the second arguments are now unified. Both are the **atom nil**, therefore this call succeeds.
7. Procedure **u/0** calls procedure **d/1** passing it **X**.
8. Procedure **d/1** will succeed if the incoming argument can be unified with the list represented by prototype #3. Again both are **list/2** structures. The first argument of **X**, which is at location 0 on the structure stack, is unified with the **atom a**. Because location 0 contains a reference cell, it is dereferenced to location 1 which is instantiated to **1: <atom, a>**. The unification procedure continues and succeeds.
9. Procedure **u/0** calls procedure **e/1** passing it **Y**.

10. Procedure **e/1** will succeed if the incoming argument can be unified with the list represented by prototype #4. Again both are **list/2** structures, and the first argument of **Y**, which is at location 1 on the structure stack, is unified with the atom **b**. Location 1, however, has been previously bound to the atom **a**, and therefore this call fails. Because there are no choice points in the program, the program fails to find a proof to the query **u**.

A.2.2 Structure Copying

The same example program is used to illustrate the construction and instantiation of variables in a structure copying implementation. In this case the execution is as follows:

1. Procedure **u/0** calls procedure **a/1** with the argument **X: <undef, undef>**.
2. Procedure **a/1** unifies the incoming argument with a list containing a single variable, causing **X** to become **X: <ref, pointer to top of copy stack which is location 0>**.

The copy stack is set up with a copy of prototype #1 as follows:

```
2:<atom,nil>
1:<undef,undef>
0:<struc,list/2>
```

3. Procedure **u/0** calls procedure **b/1** with the argument **Y: <undef, undef>**.
4. Procedure **b/1** instantiates **Y** to **Y: <ref, pointer to top of copy stack which is now location 3>**. Locations 3 through 5 of the copy stack are equivalent to locations 0 through 2 respectively.
5. Procedure **u/0** calls procedure **c/2** passing it **X** and **Y** which are now reference cells.

6. Procedure **c/2** is true if the two incoming arguments can be unified. Each argument is dereferenced to its respective copy stack structure. Using the steps given in Section 2.1.3, the functor/arity of the two structures are matched, and since both are **list/2**, the first arguments are unified. This unification causes the copy stack to be updated to:

```

5:<atom,nil>
4:<ref,pointer to location 1 of the copy stack>
3:<struc,list/2>
2:<atom,nil>
1:<undef,undef>
0:<struc,list/2>

```

Copy stack location 1 is senior to 4, therefore 4 is set to reference 1.

7. Procedure **u/0** calls procedure **d/1** passing it **X**.
8. Procedure **d/1** will succeed if the incoming argument can be unified with the list represented by prototype #3 of Figure A.3. After **X** is dereferenced, both are **list/2** structures, therefore, the first argument of **X** which is at location 1 on the copy stack, is unified with the atom **a**, and the copy stack becomes:

```

:
2:<atom,nil>
1:<atom,a>
0:<struc,list/2>

```

9. Procedure **u/0** calls procedure **e/1** passing it **Y**.
10. Procedure **e/1** will succeed if the incoming argument can be unified with the prototype in Figure A.4. After dereferencing **Y** to location 3 of the copy stack both arguments are found to be **list/2** structures, therefore, location 4 of the

copy stack is unified with the atom **b**. This fails because location 4 dereferences to location 1 which contains the atom **a**. As in the structure copying case, there are no choice points in the program which fails to prove the query **u**.

A.3 Comparison of Structure Sharing and Structure Copying

In a structure copying implementation, if a compound term is to be constructed as a result of successful unification, a copy of the prototype is made to represent the instance of the term. Therefore, all compound terms reside on the copy stack, and all variable-to-variable bindings can be made to *senior* variables, (value cells constructed earlier in the proof), as shown in Figure 2.7. Value cells on the copy stack are always considered to be senior to those which are on the *control stack*, (this is analogous to the Pascal call stack) because they have an equal or longer lifetime when compared to the clause frame variables. The fact that all structures reside on the copy stack implies that, on backtracking this stack can be restored by simply resetting the pointer to the top of the stack, to its value on entry into the nearest choice point's instance. Up to now this *environment* (the goal's instance) contained only the variable frame and two pointers: one to the parent goal which allows this goal to return correctly, and one to the next alternative in the corresponding clause list. The top of the copy stack, and a pointer into the control stack where the nearest choice point exists are two other pointers which must also be saved in an environment's state information for use during backtracking.

Structure copying also allows the clause environment, stored on the control stack, of a successfully and determinately completed clause to be deallocated before continuing with the next goal in the proof. A clause exits *determinately* if there are no alternative branches to take on backtracking, and since all relevant information can be stored on the copy stack and in global pointers, the space is reclaimable.

In a structure sharing implementation, it is possible for any value cell to reference the control stack since this is where the variable frames exist. This occurs, for example if an unbound variable introduced early in the proof is bound to a com-

pound term whose prototype and thus variable frame are part of the current goal's environment. The senior cell, the undefined cell, is assigned the following tag and value pointer:

*<pointer to prototype node in the current clause's representation,
pointer to the variable frame of the current goal>*

In this case the undefined cell has been instantiated to a structure cell which points to a variable frame at the top of the control stack; the current goal's instance is at the top of the stack. On backtracking, the cell must be unbound to avoid having a *dangling reference*; that is, a pointer to nowhere. In general, the exercise of unbinding variables when backtracking is more complicated in a structure sharing model. The variable frame of this implementation is an integral part of the clause instance which cannot be deallocated as long as the variables are accessible; i.e. until backtracking takes the execution back past this goal's invocation.

Allowing references to the control stack also implies that environments must stay in tact as long as the proof is succeeding, therefore environments of determinate clauses cannot be *trimmed* (reclaimed). In both implementations, when an undefined variable which was introduced before the nearest choice point, is set to a particular value by unification, the update must be remembered. Only undefined variables can be assigned another value, therefore to reset a variable on backtracking, the value cell is unbound. This action only requires the value cell locations. These value cell pointers are stored on the *trail stack* (cref. § 2.3.2). Because of the potential for dangling references, a structure sharing implementation will have more trail stack entries, and thus a slower clean-up process when backtracking.

Structure sharing implementations are, therefore, advantageous in applications where large or several compound terms are being constructed and little backtracking is expected, and structure copying implementations are suited to applications with smaller structures where backtracking is more likely to occur. Structure sharing can be improved by recognizing which variables have long lifetimes, and placing their frame cells on a *global* stack while the remaining variables are placed on a *local* stack. In general, although structure sharing implementations tend to run faster, the structure copying approach is simpler and reasonably memory efficient, therefore,

the later seems to be favored [13].

A.4 Effects of Execution on the Components of an Interpreter

A query execution begins with the following steps:

1. The arguments in the query call are assigned to the argument registers¹. An example of this is given in Appendix A.2. Compound term representations are constructed on the copy stack, and pointed to by reference cell arguments, therefore, the global pointer to the top of the copy stack might have a value other than zero when the first “call” is made.
2. The goal predicate symbol is a pointer into the atom table from which the prototypes of the first clause are obtained.
3. The initialization of this clause environment begins.
4. Space for the clause variable frame is allocated, and each value cell is set to: `<undef, undef>`.
5. The environment pointers are assigned as follows:
 - (a) The top of the copy stack pointer is assigned the value of the global pointer.
 - (b) The top of the trail stack pointer is assigned the value of the global pointer which is currently pointing to the bottom of the stack.
 - (c) The continuation pointer is set to point to the second subgoal in the body prototype. For a fact, or a single subgoal rule, this pointer is NULL.
 - (d) The parent environment of this first call is non-existent, therefore, this is a NULL pointer.

¹Argument cells are usually called *argument registers*, however, they are not necessarily stored in actual machine registers. They are, however, value cells.

- (e) No nearest choice point is also indicated by a NULL pointer.
 - (f) The alternative pointer is set to reference the next clause in the procedure clause list. If there are no more clauses, then this is assigned NULL which indicates that this is a determinate clause.
6. The argument registers are copied into the environment. If, however, this is a determinate clause, there is no need to perform this copy. By not copying the arguments, memory is saved.
 7. The global pointers are assigned their appropriate values:
 - (a) The top of the control stack pointer is set to reference the location following the last entry in the clause environment.
 - (b) The choice point is defined to be the same as the environments nearest choice point.
 - (c) The choice point on the copy stack is assigned the value of the global pointer to the top of this stack.

The next step is to unify the query arguments with the clause head prototype. Successful unification leads to the assignment of the arguments to be passed in the first goal call. Appendix A.2 provides an example of unification showing how argument cells are assigned, and how the local variables and copy stack are involved.

As the proof proceeds, unification causes the copy and trail stacks to grow, and with each successful goal match, the control stack grows. The only time in a forward proceeding proof, at which control stack space can be reclaimed is on a determinate clause return. When a determinate clause is ready to return to its parent, the pointer to the top of the control stack is set re-use the space taken to prove this determinate clause. In all other successful returns, (i.e. non-determinate completions), the top of the control stack comes after the environment of the last goal required to prove the current subgoal. This can be quite a distance from the parent, but marks the spot where the call to the next subgoal must begin. Determinate exits, (returns where the control stack space is reclaimed), are the main selling point to the structure copying implementation.

If a unification fails, the program backtracks triggering the following events:

1. The global pointer to the top of the control stack is set equal to the current choice point.
2. If the last alternative of this clause has been reached, then the global choice point is set to this environment's nearest choice point.
3. The global pointer to the top of the structure stack is set to the local pointer of the new environment.
4. Locations referenced by the trail stack from the local top pointer to the global top pointer are reset to undefined.
5. The global pointer to the top of the trail stack is set to the local pointer of the new environment.
6. The current environment is reset to reflect the alternative clause.

A.5 An Optimizing Feature of Prolog

Tail recursion optimization (TRO) is used to reduce the memory demands of Prolog execution. In a structure copying implementation, it was noted, that a determinate exit from a clause environment means that the corresponding control stack space can be reclaimed. A determinate exit was defined to be a successful completion of the clause proof where no alternatives to that clause exist. A clause is tail recursive if the last subgoal of the clause body has the same functor/arity as the head goal. If a clause is tail recursive, determinate, and has not created any choice points before the last subgoal is to be called, then it will execute iteratively. In this situation, the same clause environment on the control stack can be used for each iteration of the execution, therefore the effort of repeatedly allocating and deallocating this environment is no longer required.

There are two easily recognized clause formats which have the potential to be optimized in this way. These clause forms are:

```

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

```

Figure A.5: `member/2` predicate.

```

recursive_1 :- call_1, ..., call_n, !, recursive_1.

```

and

```

recursive_2 :- recursive_2.

```

In both cases the clauses are tail recursive, and have not introduced any choice points. The first example ensures that there are no choice points by adding a `cut` operator before the last call, the second example only contains the recursive call. If these clauses are determinate, then tail recursion optimization may be used during their execution. For example, the last clause of the `member/2` procedure given in Figure A.5 can be optimized this way because:

1. The clause is tail recursive.
2. The last clause is determinate.
3. No choice points are created before the recursive call.

A.6 WAM Unification Instructions

A.6.1 Get Instructions

The set of WAM match instructions, which are used to unify the head goal terms with the contents of the argument registers are:

- `get_const(constant, argument register)` for unifying with a constant term,
- `get_variable(local variable location, argument register)` for unifying with the first occurrence of a variable,

- **get_value**(*local variable location, argument register*) for unifying with the second or later occurrence of a variable. This requires the general unification algorithm,
- **get_structure**(*functor/arity, argument register*) for starting unification with a compound term,
- **get_nil**(*argument register*) which is a special case of **get_const** used for the empty list atom, and
- **get_list**(*argument register*) which is a special case of **get_structure** used for lists.

Local variable locations are either: (1) on the control stack in the case of permanent variables, or (2) in free argument registers² for temporary variables.

A.6.2 Unify Instructions

A **get_structure** instruction is followed by a list of unify instructions, one for each argument of the compound term. Nested structures are constructed and referenced through temporary variables much like the example (cref. § 2.2.2) which showed the construction of a compound term with the inner structures built after the outer ones. The instructions work at an “indicated location” on the copy stack which is determined by the **get_structure** or **put_structure** instruction. A **get_structure** instruction is passed the local variable value cell, which represents either an unbound variable or a reference to a structure on the copy stack. If the variable is a reference, then the arguments of the structure have been allocated and reside immediately above the tag field which is being referenced. The “indicated location” is one location above the referenced location, and the unify instructions are to *match* these arguments with the structure of the clause. If the local variable is still undefined, then a *copy* of the clause structure must be built at the top of the copy stack. When matching, the unify instructions are said to be in *read* mode: when copying, they are in *write* mode. The **get_structure** instruction provides

²The first free argument register is indexed by one more than the highest arity subgoal.

a location at which the *unify* instructions are to operate, and the mode, (*read* or *write*), in which they are to work.

In the body of the clause, a structure is being passed as an argument. In this situation, a copy of the term must be built on the copy stack. Therefore, the "indicated location" is always the top of the stack, and the *unify* instructions always work in *write* mode. For example, step (4) of the example given earlier in this section stated that a series of *argument* and *unify* instructions would be required to build the *blue/2* structure on the copy stack. Those instructions are listed below, and the description of all *unify* instructions follows them:

- (4a) `put_structure(blue/2, argument register 2)`
- (4b) The arguments of *blue/2* are now unified.
`unify_value(local variable location of X)`
- (4c) `unify_variable(the next free temporary location = argument register 7)`
 Nested structures are built and referenced by temporary variable locations, therefore, argument register seven contains a reference to the *mix/2* structure.
- (4d) Build the *mix/2* structure
`put_structure(mix/2, argument register 7)`
- (4e) The arguments of *mix/2* are now unified.
`unify_variable(the next free temporary location = argument register 8)`
- (4f) `unify_constant(b)`
- (4g) Build the *cons/2* structure which is being referenced by the eighth argument register.
`put_structure(cons/2, argument register 8)`
- (4h) The arguments of *cons/2* are now unified.
`unify_value(local variable location of X)`
- (4i) `unify_constant(a)`

The *unify* instructions are described below. Each provides the action to be taken in *read* mode, followed by the action taken in *write* mode.

- `unify_const(constant)` which specifically unifies with, or constructs a constant at, the indicated location on the copy stack.

- **unify_variable**(*local variable location*) which binds the local variable to, or constructs an unbound variable at, the indicated location on the copy stack.
- **unify_value**(*local variable location*) which performs a general unification with, or pushes the value of the local location onto, the indicated location on the copy stack.
- **unify_void**(*number of void terms - N*) which is used for single occurrence variables, and will either skip matching with the next N arguments of the copy stack, or push N unbound variables on the copy stack. This represents optimization 3 of the list of unification simplifications given earlier in this section.
- **unify_local_value**(*local variable location*) which is like **unify_value** but ensures that the copy stack location is being used as a more senior location to any on the control stack.

A.6.3 Put Instructions

The argument instructions, put instructions, are similar to the two cases mentioned above, but are used to assign values to the argument registers passed in the next call. These instructions represent the terms in the clause body. The **put_structure** instruction will always set the unify instructions which follow into write mode. The specific instructions are:

- **put_const**(*constant, argument register*) for simply assigning a constant to the argument register,
- **put_variable**(*local variable location, variable classification, argument register*) If the variable is temporary an unbound variable is created on the copy stack and referenced by the argument register and the temporary variable. For permanent variables, the local location is set to be an undefined variable, and this is referenced by the argument register,

- **put_value**(*local variable location, argument register*) which simply moves the contents of the local variable into the argument register,
- **put_structure**(*functor/arity, argument register*) for starting the construction of a structure which is referenced by the argument register,
- **put_nil**(*argument register*) which is a special case of **put_const** used for the empty list atom,
- **put_list**(*argument register*) which is a special case of **put_structure** used for lists. and
- **put_unsafe_value**(*permanent variable location, argument register*) which “globalizes” the last occurrence of an unsafe variable if necessary, and otherwise acts like **put_value**.

A.7 WAM Control Instructions

The WAM compiler will distinguish between facts, single subgoal rules (e.g. *a :- b.*), and multiple subgoal rules (e.g. *a :- b,c,d.*) which each use different control instructions in order to proceed with the proof.

facts Facts and single subgoal rules do not create environments in the WAM implementation because all variables are temporary. This implies that there are no value cells needed in the environment, and all relevant information for the proofs execution can be stored in the global pointers. If a fact matches successfully with the arguments passed in the call, it *proceeds* by setting the *program pointer*, which points into the code space, to the current *global continuation pointer*, which points into the code space at the next instruction to be executed. For facts, the *match* and *unify* instructions which unify the arguments with the head goal terms, are followed by a *proceed* instruction as shown in Figure A.6.

multiple subgoal rules A rule with multiple subgoals does require a control stack environment. The contents of which are quite similar to that discussed in § 2.3.

```

    get args of head goal
  proceed

```

Figure A.6: WAM instruction format for a fact.

The code begins with an *allocate* instruction which sets up the environment: it allocates the space, assigns the appropriate values to the local pointers, and creates an unbound value cell for each permanent variable. This is followed by a series of *match* and *unify* instructions, and the *argument* and *unify* instructions of the first subgoal call. As was shown in the example at the beginning of this section, a *call* instruction is next. It assigns the next instruction of this clause, (which is the first *argument* instruction of the next subgoal call), to the continuation pointer, and sets the program pointer to the first instruction of the subgoal's code. When this goal call returns, the program pointer is set to the continuation pointer which starts the put instructions for the next subgoal. This cycle continues until the put instructions of the last subgoal are complete. At this point, the originally saved continuation pointer, (which points to the next instruction to be executed by the clause's parent), is restored, and the current environment is discarded by a *deallocate* instruction. The last subgoal is *executed*. The **execute** instruction also assigns the subgoal's first instruction location to the program pointer; but, instead of returning to the parent on completion of the subgoal's code, the proof continues at the instruction referenced by the continuation pointer. For a clause with *n* subgoals, code with the pattern shown in Figure A.7 would be generated:

single subgoal rules A single subgoal rule is a special case of the multiple subgoal rule. As mentioned, no environment is required, therefore this is equivalent to the last goal call of the more general case; that is, an **execute** instruction is used to invoke the subgoal of this clause as shown in Figure A.8.

```

allocate space for environment
get args of head subgoal
put args of first subgoal
call the first subgoal
:
: put args of  $i_{th}$  subgoal and call the same
:
put args of  $n_{th}$  subgoal
deallocate the environment
execute the last subgoal

```

Figure A.7: WAM instruction format for a multiple subgoal rule.

```

get args of head goal
put args of first and only subgoal
execute the only subgoal

```

Figure A.8: WAM instruction format for a single subgoal rule.

A.8 WAM Indexing Instructions

The result of **try**, **retry**, and **trust** instructions are as follows:

try The **try** instructions create choice points which are saved by establishing an environment on the top of the control stack for each alternative; that is, the current environment is not necessarily at the top of the stack. Choice point environments require much more control information including: (1) copies of the argument registers, which are used destructively by the **get** and **put** instructions, (2) a current environment pointer, (3) a previous choice point pointer, (4) the current continuation pointer, (5) a pointer to this alternative's code, and (6) the pointers to the top of the trail and copy stacks.

retry Retry instructions indicate that an intermediate choice point, for which the environment already exists, is to be considered next.

trust Trust instructions imply that this is the last alternative of the procedure. Global pointers are reset to the values stored by the parent clause; i.e. pointers stored in the parent's environment.

A.9 WAM Indexing Instructions — Example

Figure A.9 provides the Prolog definition for **concatenate/3**. The first term of the first clause is the atom **nil**; the first term of the second clause is a list. The **switch_on** instruction, which is followed by four arguments which indicate the instruction label to jump to in case of: (1) a variable, (2) a constant, (3) a list, or (4) a structure, would be:

switch_on_term

- (1) *location of an instruction which indicates that both clauses should be tried,*
- (2) *location of the match instruction which represents the atom nil (i.e., the first clause),*
- (3) *location of the match instruction which starts unification of the*

```
concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).
```

Figure A.9: `concatenate/3` predicate.

list in clause 2,

- (4) **fail**, which implies that the procedure fails if the first argument is a structure.

The complete set of WAM instructions for the `concatenate/3` procedure are given in Figure A.10 [30]. The `switch_on` instruction determines the type of the first argument and jumps according to its type to the instruction defined by the supplied label as follows:

variable -> **C1a** — When the first argument is a variable, the `switch_on` instruction sends the execution to the instruction at the label **C1a** which is `try_me_else C2a`. This creates a choice point environment for the second clause before continuing with the next instruction. The instruction at **C2a** is marked as the *continuation* instruction in case of a failure during the proof of the first clause. At **C2a** there is a `trust_me_else fail` instruction. This instruction indicates that the code which follows is for the last alternative of the procedure, the global pointers are reset to return to the parent clause on completion of this goal proof.

constant -> **C1** — Label **C1** is not preceded by any instruction which creates choice points or resets pointers. The `proceed` instruction of the code which follows, will return to the parent clause immediately, as would a failure during the *match* and *unify* instructions. When the first argument is a constant, no effort is wasted on the second clause because it would never unify successfully.

list -> **C2** — At label **C2** the code executes as a single clause procedure. When a list is passed as the first argument, there is no point trying to unify the calling goal with the first clause, therefore, this code is skipped over.

```

concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :- concatenate(L1,L2,L3).

concatenate/3:
    switch_on_term C1a,C1,C2,fail % jump if variable, constant,
                                % list, or structure
C1a:      try_me_else C2a      % concatenate(
C1:      get_nil A1           % [],
                                % omit an unnecessary
                                get_variable A2,A2
                                % L,L
                                proceed % ).
C2a:      trust_me_else fail   % concatenate(
C2:      get_list A1          % [
                                unify_variable X4 % X|
                                unify_variable A1 % L1], L2
                                % omit an unnecessary
                                get_variable A2,A2
                                % [
                                unify_value X4 % X|
                                unify_variable A3 % L3]) :-
                                % omit an unnecessary
                                put_value A1,A1
                                % omit an unnecessary
                                put_value A2,A2
                                % omit an unnecessary
                                put_value A3,A3
                                execute concatenate/3 % concatenate(L1,L2,L3).

```

Figure A.10: Example of WAM compiler output for the *concatenate* procedure.

structure -> fail — A structure would not unify with either clause, therefore the procedure returns immediately to the nearest choice point.

In this example, **Ai** represents argument register *i*, and **Xi** represents temporary variable location *i* which is equivalent to the corresponding argument register. The example shows how the compiler chooses temporary variable offsets such that the maximum number of null instructions are introduced; note that the variable **L1** of the second clause head goal is unified with argument register **A1** in order to eliminate having a put instruction before the execute instruction. Several more examples can be found in [30].

Appendix B

Selected 'C' Support Software

This appendix contains selected 'C' software which supports the execution of the PIC procedures.

B.1 manage_action

```
/******
```

```
Function:      action manage_action(
                continuation_pointer,
                copy_stack,
                trail_stack,
                nearest_copy_stack_choice_point );
```

- This function calls the next function in the continuation and reacts according to its return. A determinate return invokes the goal of the next continuation closure, and a success implies that the current branch is complete.

```
*****/
```

```

action manage_action (cont,global_stack,trail_stack,choice_point )
contin      cont;
g_stack     global_stack;
t_stack     trail_stack;
int         choice_point;
{
    contin      orlist1, orlist2;
    int         or_global_top, or_trail_top;
    action      next_action;
    next_action = determinate;
    while ( next_action == determinate ) {
        if ( or_split( cont ) == true ) {
            make_or_lists( cont, &orlist1, &orlist2 );
            or_global_top = global_stack->top;
            or_trail_top = trail_stack->top;
            next_action = manage_action(orlist1,global_stack,
                                      trail_stack,choice_point);
            if ( next_action == fail ) {
                clean_up(global_stack,or_global_top,
                        trail_stack,or_trail_top);
                next_action = manage_action(orlist2,global_stack,
                                      trail_stack,choice_point);
            }
            remove_rem( &orlist1, cont->elem.orlist.rem );
            remove_rem( &orlist2, cont->elem.orlist.rem );
        }
        else {
            if ( cont == NIL )
                next_action = success;
            else {
                /* call the next function in the continuation until fail

```

```

        is returned or the continuation is empty (success is
        returned) */
        next_action = ((*cont->elem.func.ptr))
                        ((cont->elem.func.args),cont->elem.func.next,
                        global_stack, trail_stack, choice_point);
        cont = cont->elem.func.next; /* pointer changed only
                                      within this scope */
    }
}
}
if ( next_action == fail )
    return(fail);
else
    return(success);
}

```

B.2 Differences Between WAM and PIC Unification Instructions

(1)	A new instruction put_unsafe has been added to create a globalized unbound instance of an unsafe variable on its first occurrence.
(2)	The instructions unify_local_value and put_unsafe_value dealt with the last occurrence of an unsafe variable in WAM. Because of the immediate globalization of these terms in PIC, these instructions are not required.
(3)	The special case instructions used for lists and the empty list constant are not used in this implementation because their representation is consistent with all structures and atoms respectively, therefore, there is no advantage is separating out these cases.
(4)	The unify_void instruction is not used because: <ol style="list-style-type: none"> 1. The translator does not look ahead to the next term, which it would have to do to determine N, the number of void terms to repeat. 2. The global variable unify_add, which is initialized by the get_struc and put_struc instructions as the pointer to the first structure argument, must be updated (incremented) after each term is considered which is basically all a unify_var instruction does anyway.
(5)	The get_const and put_const instructions are each split into four instructions: get or put_int for integers, _real for floating point numbers, _str for atoms, and _quoted for character strings.
(6)	put_variable is split into put_temp_var (temporary) and put_perm_var (permanent) because there was no common component to these functions.
(7)	The parameter lists are extended, when necessary, to include the copy and trail stack pointers, and the nearest choice point's copy stack pointer which indicates when trailing is required.

Table B.1: Differences between WAM and PIC **get**, **put**, and **unify** instructions

Appendix C

Detailed Execution Example

This appendix will show, in detail, the execution of a PIC function for the `member/2` predicate shown in Figure 4.1. The generated output is given in Figures C.1, C.2, C.3, C.4, and C.5.

C.1 Example 1

If `member2` is declared as:

```
member2( int:in, int_list:in );
```

where `int_list` defines a list of integers as explained in Section 4.1 of Chapter 4, and if it is called with:

```
member(3, the 'C' structure equivalent to: [1,3,5]);
```

then the following steps occur during the proof of this query:

1. The CPI function passes the following arguments in the call to `member2`:
 - (a) An argument list where the first argument represents the integer 3, and the second argument defines the list [1,3,5]. The argument stack and copy stack assignments for these parameters are shown in Figure C.6.
 - (b) A pointer to the top and bottom of the copy stack. The top, when the call is made, is at location 15 as shown in Figure C.6.

```

/*****
FILE :          member.c
PROLOG to C translator output          */
#include "pic.ty"
#include "pic.var"
#include "pic.ext"
#include "expand.c"
/*****/
action member2( arg_list, continuation, global_stack,
                trail_stack, choice_point )
arg_list_type  arg_list;
contin         continuation;
g_stack        global_stack;
t_stack        trail_stack;
int            choice_point;
{
    contin      my_cont, cur_cont;
    action      next_action;
    boolean     determ, continu, bound, saved;
    int         backtrack_level, global_top, trail_top,
                new_choice_pt, 1;

    Atom        arg1_type;
    arg_list_type arg_list_1;
    arg_list_type save_args;
    mode_type    mode;
    obj_ptr      unify_add;

```

Figure C.1: Function declaration and local variables of the `member2` PIC function.

```

global_top = global_stack->top; /* important for clean up */
trail_top  = trail_stack->top;
continuu = true;
backtrack_level = 0;
saved = false;
/* Determine the argument type of the first incoming argument. */
arg1_type = (Atom)(*(dereference(arg_list, &bound)));

while (continuu==true)
{
    switch (backtrack_level)
    {
        case 0:
            determ = false; /* first arg_type switch not necessary */
            new_choice_pt = global_stack->top;
            GET_STRUC( list_2, &arg_list[2*1], &mode, &unify_add,
                      global_stack, trail_stack, new_choice_pt );
            UNIFY_VALUE( &arg_list[2*0], &mode, &unify_add,
                        global_stack, trail_stack, new_choice_pt );
            UNIFY_VARIABLE( &arg_list[2*2], &mode, &unify_add,
                           global_stack );
            mode = write;
            my_cont = NIL;
            cur_cont = continuation;
            next_action = manage_action(cur_cont, global_stack,
                                       trail_stack, new_choice_pt);
            delete_cont(&my_cont, continuation);
            break;
    }
}

```

Figure C.2: First clause of the `member2` PIC function.

```

case 1:
    determ = true; /* last clause */
    new_choice_pt = choice_point;
    GET_STRUC( list_2, &arg_list[2*1], &mode, &unify_add,
               global_stack, trail_stack, new_choice_pt );
    UNIFY_VARIABLE( &arg_list[2*2], &mode, &unify_add,
                    global_stack );
    UNIFY_VARIABLE( &arg_list[2*3], &mode, &unify_add,
                    global_stack );
    mode = write;
    put_value( &arg_list[2*0] , &arg_list_1[2*0] );
    put_value( &arg_list[2*3] , &arg_list_1[2*1] );
    if ( saved == false )
    {
        saved = true;
        for (i=0; i< arglistlen; i++)
            save_args[i] = arg_list[i];
    }
    for (i=0; i< arglistlen; i++)
        arg_list[i] = arg_list_1[i];
    arg1_type = (Atom)(*(dereference(arg_list, &bound)));
    goto tail_recursive;
break;

default:
    next_action = fail;
    continu = false;
    break;
}

```

Figure C.3: Second clause of the `member2` PIC function.

```

switch ( next_action )
{
    case fail:
        if ( determ == true )
        {
            continu = false;
            if ( saved == true )
                for (i=0; i<arglistlen; i++)
                    arg_list[i] = save_args[i];
        }
        else
        {
            clean_up(global_stack, global_top,
                    trail_stack, trail_top);
            ++backtrack_level;
        }
        break;
    case determinate:
    case success:
        continu = false;
        break;
}
goto loop;

```

Figure C.4: Switch block of the `member2` PIC function to control execution according to the `next_action` status returned by `manage_action`.

```

/* GOTO labels */
failed: if ( determ == true )
{
    next_action = fail;
    continu = false;
    if ( saved == true )
        for (i=0; i<arglistlen; i++)
            arg_list[i] = save_args[i];
}
else
{
    clean_up(global_stack, global_top,
            trail_stack, trail_top);
    ++backtrack_level;
}
goto loop;
shallowfail: /* no stack clean_up is required */
    next_action = fail;
    ++backtrack_level;
    goto loop;
tail_recursive:
    global_top = global_stack->top;
    trail_top = trail_stack->top;
    backtrack_level = 0;
    goto loop;
loop: ;
}
return(next_action);
}

```

Figure C.5: Goto labels of the `member2` PIC function.

- (c) A pointer to the top and bottom of the trail stack which is location 0 in both cases.
 - (d) A NULL pointer is passed as the continuation.
 - (e) The copy stack choice point which is passed is equal to zero: i.e. the zeroth location of the array.
2. The **member2** PIC function begins by saving: (1) the current index to the top of the copy stack called the **global** stack in this implementation, and (2) the current index to the top of the trail stack. The next few lines of Figure C.2 initialize local flags and counters.
 3. **arg1.type** is assigned the tag type of the first argument: the value cell must be dereferenced in order to obtain this information. Clause indexing is not used in this procedure because the first term of both clauses is a variable, therefore, this line could be removed when tuning.
 4. The first clause is tried. It is non-determinate and, therefore, the choice point is updated to the current index for the top of the global stack.
 5. The first occurrence of **X** (Figure 4.1) does not require a **get.var** instruction because this is a temporary variable, therefore, this argument register is already a valid location.
 6. The second argument (**arg_list[2*1]**) is unified with the structure **list_2**. The **2** in **2*1** represents the number of locations in a value cell. The **1** defines the argument number which start with zero. The statement **GET_STRUC(x)** is expanded into:

```
if ( get.struc(x) == false ) goto failed,
```

by the macro expansion feature of the 'C' programming language. This line dereferences the contents of **arg_list[2*1]** to location 0 of the global stack as shown in Figure C.6. Because this location contains the tag which indicates the start of a **list_2** structure, the call succeeds and assigns a pointer to

Figure C.6: Argument mini stack and copy stack passed in the '(C) member2 call. member2(3, [1, 3, 5]).

the next location - 1, to the variable **unify_add**. This parameter defines the "indicated location" used by the unify instructions to find the current structure argument.

7. **unify_value** is passed a pointer to the first incoming argument, **X**, and unifies it with the term at **unify_add**. A call to the general purpose unification function is made, but it returns **false** because, although both terms are integers, the values (3 and 1) do not match.
8. The execution jumps to the **failed** label of Figure C.5.
9. Here **cleanup** is called to reset the copy and trail stacks. However, because no bindings have occurred, this does not actually accomplish anything.
10. **backtrack_level** is incremented to 1, and the execution loops around to the second clause at **case 1** of Figure C.3.
11. The last clause of a procedure is determinate.
12. As with the first clause, the first argument does not require a **get** instruction; i.e. the instruction to move the first argument to a temporary location is not necessary since it is already in one.
13. The call to **get_struc** achieves the same result as it did for the first clause.
14. **unify_variable** is called next and passed the first free, argument stack location which represents the first available temporary variable location. This creates a new reference cell whose value pointer is assigned the value of **unify_add**; i.e. a reference to location 1 of the global stack. The value of **unify_add** is updated so that it points to the next structure argument which is two locations above the current; i.e. location 3.
15. The next call to **unify_variable** creates a reference cell in the next available temporary location which points to location 3 of the global stack. The argument stack at this point is shown in Figure C.7.

<i>pointer to global stack location 3</i>
<code>^ ref</code>
<i>pointer to global stack location 1</i>
<code>^ ref</code>
<i>pointer to global stack location 0</i>
<code>^ ref</code>
<code>^ 3</code>
<code>^ integer</code>

Figure C.7: Argument mini stack after the `get` instructions of the first call are completed.

16. The `put_value` instructions copy the contents of the first incoming argument and the second temporary variable into the argument mini stack which is to be passed in the call to the first subgoal; that is, `arg_list.1`. This argument stack is shown in Figure C.8.
17. This clause is both tail recursive and determinate, therefore it can be optimized with TRO. In this situation, the original arguments are saved, and the arguments of the recursive call are copied over to the argument stack of this procedure.
18. The next line in the code could also be removed by tuning. It simply resets the `arg1_type` variable used for clause indexing the new arguments.
19. The execution then jumps to the label `tail_recursive`. Here, the variable `backtrack_level` is reset to zero, and the code loops through the `while` block instead of making a procedure call.

<i>pointer to global stack location 3</i>
<code>^ ref</code>
<code>^ 3</code>
<code>^ integer</code>

Figure C.8: Argument mini stack to be passed to the first subgoal of the second clause. TRO is not being considered yet.

20. This time around, **get_struc** of the first clause is looking for a **list_2** structure at the new second argument. After dereferencing this argument, a matching structure is found at location 5 of the global stack. The variable **unify_add** is set to point to the next location which is location 6.
21. **unify_value** must now match the integer 3, with the term at location 6. This succeeds.
22. The local continuation for a fact is represented by a **NULL** pointer, and the continuation list passed in is also a **NULL** pointer, therefore **manage_action** returns the status **success** with little effort.
23. The function **delete_cont** frees the locally allocated continuation list elements, which, in this case, requires no work.
24. The execution breaks out of the **switch** block for the clause alternatives, and begins at the next **switch** construct shown in Figure C.4.
25. The **contin** flag is updated to cause the execution to leave the **while** loop: **next_action** remains equal to **success**.
26. The last statement executed is the last line of Figure C.5. It returns **success** to the calling function which is the CPI function for the C-Prolog call.

Appendix D

Built In Predicates

This section lists the built-in PIC predicates provided with the development system. For the most part, the names and functionality are consistent with MProlog.

D.1 Control Built-in Predicates

The predicates discussed in this section affect the control of the execution, they are:

!/0 — The cut operator removes the choice points defined by subgoals to the left of the cut in a particular clause, and the choice points of the parent goal.

‘;’/2 — The **or/2** predicate allows a disjunction in a clause.

fail/0 — The **fail** predicate causes a determinate failure to occur.

succeed/0 — The **succeed** predicate represents a determinate success.

evaluate/1 — This predicate supports meta-evaluation. (ref. § 3.3.5).

not/1 — **not/1** supports negative logic (ref. § 3.3.5).

repeat/0 — The **repeat** predicate always succeeds the first time it is called, and every time it is re-evaluated due to backtracking.

D.2 Arithmetic Built-in Predicates

In this section, an arithmetic statement is a ground sequence of integer and real numbers linked together by any of the following operators:

$+$ for addition,
 $-$ for subtraction,
 $*$ for multiplication,
 $/$ for division, and
mod for modulus.

For example:

$3+(5-2)/3$

is a ground arithmetic statement which represents the value **4**. The available arithmetic operations are:

is/2 This predicate assigns the value of the arithmetic statement passed as the second argument to the unbound variable passed as the first argument; e.g. **X is 3+(5-2)/3** will bind **X** to: **<int,4>**. If **X** does not dereference to an unbound variable when this predicate is called, the procedure will return a **fail** status.

relational operations The relational operators can be used in infix notation. They accept two ground, arithmetic statements and succeed only if the condition is true. The relations offered are:

$'>'/2$ e.g is **X** greater than **Y**, is written **X > Y**.
 $'>=' /2$ e.g is **X** greater than or equal to **Y**, is written **X >= Y**.
 $'<'/2$ e.g is **X** less than **Y**, is written **X < Y**.
 $'<=' /2$ e.g is **X** less than or equal to **Y**, is written **X <= Y**.
 $'=='/2$ e.g is **X** equal to **Y**, is written **X == Y**.
 $'/= '/2$ e.g are **X** and **Y**, not equal is written **X /= Y**.

D.3 Dynamic Operations

This section will list the predicates which operate on dynamic clauses.

add_statement/1 — This predicate will assert its argument at the end of the corresponding clause definition.

add_statement_a/1 — This predicate will assert its argument at the beginning of the corresponding clause definition.

del_statement/1 — This predicate will retract the first clause to match its argument from the corresponding dynamic clause list.

add_statement_b/1 — This predicate represents a backtrackable version of **add_statement/1**. It is equivalent to:

```
add_statement_b(X) :- add_statement(X).
add_statement_b(X) :- del_statement(X), fail.
```

del_statement_b/1 — This predicate represents a backtrackable version of **del_statement/1**. It is equivalent to:

```
del_statement_b(X) :- del_statement(X).
del_statement_b(X) :- add_statement(X), fail.
```

spy_dyn/1 — This predicate matches its argument with the clauses of the corresponding dynamic clause list. It is automatically added by the translation process to all goal calls of predicates declared as dynamic.

definition_length/2 — This predicate counts the number of clauses in the dynamic predicate defined by the first argument, and unifies this with the second argument.

set_value2 — This predicate allows a value to be assigned to a *value* variable. It is used as follows: **set_value(value name, value)**.

value/2 — This predicate unifies the second argument with the value of a *value* variable. It is used similarly to **set_value**.

D.4 Input/Output Built-in Predicates

This section will describe the I/O predicates made available with the system.

set_channel/2 -- Both arguments of this predicate must be ground. If the first argument is **infile**, then the second argument, which must be a string or a quoted string, is assigned as the name of the current input stream. If the first argument is **outfile**, then the second argument becomes the name of the current output stream. No other combination of arguments is accepted.

set_input/0 -- This predicate opens the current input stream.

close_input/0 -- This predicate closes the current input stream.

set_output/0 -- This predicate opens the current output stream.

close_output/0 -- This predicate closes the current output stream.

nl/0 -- This predicate writes a new-line character to the current output stream.

nl/1 -- This predicate accepts an integer, **n**, as its argument, and writes **n** new-line characters to the current output stream.

write_tab/1 -- This predicate accepts an integer, **n**, as its argument, and writes **n** space characters to the current output stream.

write/1 -- This predicate will print the value of its argument to the current output stream. Unbound variables cause "unbound var" to be written, and structures are written strictly in prefix notation.

read/1 -- This is a non-backtrackable read which reads everything up to the first '.' that is encountered in the current input stream.

read_char/1 -- This predicate accepts three different arguments:

1. **file end** -- to be used when checking for an end-of-file character.
2. **expression end** -- to be used when checking for an end-of-statement character; i.e., '.'.

3. **line end** — to be used when checking for an end-of-line character; i.e., carriage return.

The next character will be read from the current input stream, and compared to the input character type. A success implies a match occurred. This is a backtrackable predicate in that the character read in the first clause alternative is returned to the input stream on failure in the second clause, however, the second clause always fails.

cut_input/0 — This predicate is used in conjunction with **read_char/1**. It will cause the character read in the last call to **read_char/1** to be thrown out, therefore, nothing is returned to the input stream on backtracking into the predicate **read_char/1**.

D.5 Testing Predicates

This section lists the predicates which can be used to test the type of a variable, the relative value of an integer or character string, and whether or not a variable is bound.

is_a/2 — This predicate is typically used in infix notation (e.g. *X is a type*), to test the type of a variable. The value of *type* may be any one of the following constants: **integer**, **string**, **var** (for an unbound variable), **real**, **constant**, **struc** (for a compound term), or **number** (for either an integer or real number). If the *type* argument is unbound when the procedure is called, then it will be bound to a **string** cell with one of the values listed above.

compare/3 — This predicate is used as follows:

$$\text{compare} \left(\text{relation}, \left\{ \begin{array}{c} \text{string} \\ \text{quoted} \\ \text{integer} \end{array} \right\}, \left\{ \begin{array}{c} \text{string} \\ \text{quoted} \\ \text{integer} \end{array} \right\} \right)$$

The last two arguments must be ground, and they must be the same type (i.e., **string**, **quoted**, or **integer**). The argument *relation* is one of:

$>$ - *greater than*,
 $>=$ *greater than or equal*,
 $<$ *less than*,
 $<=$ *less than or equal*,
 $=$ - *equal*, or
 $\neq$ *not equal*.

If *relation* is unbound when this function is called, it will be bound to one of: $>$, $<$, or $=$.

'=' / 2 This predicate can be used in infix notation, and it will unify its two arguments.

'\=' / 2 This predicate can be used in infix notation, and it succeeds if the two arguments cannot be unified.

'==' / 2 This predicate determines if the two arguments are *identical*; i.e., if the two arguments dereference to the same value cell. It can be used in infix notation.

'\==' / 2 This predicate will determine if its two arguments are not identical, and it may also be used in infix notation.

structure / 3 — This predicate is used as follows:

structure(*predicate*,*list*,*predicate as a list*). Either the first or last two arguments must be bound when the call is made, and the third is unified with the predicate representation or corresponding list representation accordingly. The first element of the *predicate as a list* is the predicate functor/arity symbol, and the remaining elements are the arguments of the predicate.

list_length / 2 This predicate is defined as:

```
list_length([],0).
list_length([X|T],N) :- list_length(T,N2), N is N2 + 1.
```

D.6 Character Manipulation

This section lists the predicates which manipulate character strings:

concatenate/3 — This predicate concatenates the first two arguments which must be ground string terms, and unifies the third argument with the result.

make_char_list/2 — This predicate takes a ground string term as the first argument, and generates a list structure where each element is a character in the input string. The list is unified with the second argument.

Appendix E

Testing

E.1 Using the PIC Generator

A Prolog program is translated to PIC code by taking the following steps:

1. Enter the PDSS environment and consult all of the files of the PICG¹.
2. Invoke the translator with the command: `pictrans`.
3. The translator will query the user on whether or not it is allowed to make simplifying assumptions about deterministic clauses. Answering `n`, for no, will cause the translator to not make these assumptions.
4. The translator will ask for the file names in which the program is contained. Each file contains one module of the program. All modules of the program must be compiled together because the atom table entries are initialized sequentially according to the atoms found during the translation. This sequence must match the definition of the enumerator type which is used to access the atom table; i.e, the first entry in the atom table is indexed by the first identifier in this enumerator type.
5. As the translator processes each predicate, the user is notified that the translation has completed. A file is produced for each file input, and the name

¹The files are consulted automatically according to a `pdss.init` file

of the file is the same as the name given to the program module with a `.c` extension.

6. The translator will output another file called `init_atoms.c` which contains the code to initialize the atom table. This code corresponds to the enumerator type definition found in the file `pic_args.ty`.
7. The translator also produces a file called `pic_ref.ext` which contains the external declarations (forward references) of the PIC procedures.
8. The translator produces a UNIX **make** file which pulls in all of the translated files as well as the support files such as those which contain the unification functions.
9. When the translator completes the user may exit the PDSS environment.
10. At this point, the target can be built if a mainline 'C' function and any necessary CPI functions are added to the link portion of the target **make**. The minimum required addition for a zero-arity root goal is given in Figure E.2

E.2 Benchmark Test

The naive reverse benchmark program is as follows:

```
module nrev.
/*      nrev.pro -- Naive reverse benchmark.
% *
% * This is the standard benchmark used to compute LIPS for a Prolog
% *      system. Lists should have length of at least 30 for accurate
% *      measurements. The number of iterations should be such that
% *      the benchmark runs for at least a second (the longer it runs,
% *      the more accurate the number that comes out). This program
% *      should work with all Prologs compatible with C-Prolog.
% *
% * Usage:
% * ?- reconsult(nrev).
```

3 of/de 3

MicroD

```

% * ?- nrev.
% */
body.

nrev :- write("Length of List: "),
        read(Length), read_token(expression end),
        write("Number of Iterations: "),
        read(Iter), read_token(expression end),
        genlst(Length,List),
        time(Time1), runtest(Iter,List), time(Time2),
        runcontrol(Iter,List), time(Time3),
        write(Time1), nl, write(Time2), nl, write(Time3), nl,
        compute_average_time(Time3,Time2,Time1,Iter,Avgtime),
        LI is ((Length+1) * (Length+2)) / 2 + 1,
        LIPS is (LI/Avgtime)*1000,
        write("LIPS ="), write(LIPS), nl.

runtest(N,List) :- for(1,N,_), runtestonce(List), fail.
runtest(_,_) .

runcontrol(N,List) :- for(1,N,_), runcontrolonce(List), fail.
runcontrol(_,_) .

runtestonce(List) :- nrev(List,_), !.

runcontrolonce(List) :- control(List,_), !.

nrev([],[]).
nrev([H|T],L) :- nrev(T,L1), append(L1,[H],L).

append([],X,X).
append([H|T],X,[H|T1]) :- append(T,X,T1).

control(_,_) .

compute_average_time(EndTime,MidTime,StartTime,Iter,AverageTime) :-
    AverageTime is ((MidTime-StartTime)-(EndTime-MidTime))/Iter.

```

```

for(I,I,I) :- !.
for(I,J,I).
for(I,J,K) :- I1 is I+1, for(I1,J,K).

genlst(0,[]) :- !.
genlst(N,[N|X]) :- N1 is N-1, genlst(N1,X).

time(Time) :- Time is cputime.
endmod.

```

E.2.1 Append Optimizations

This section includes the 'C' code for the `append/3` predicate before and after manual optimization.

PICG output for `append/3`

```

action append3(arg_list,continuation,global_stack,trail_stack,choice_point)
arg_list_type   arg_list;
contin          continuation;
g_stack         global_stack;
t_stack         trail_stack;
int             choice_point;
{
    contin       my_cont,cur_cont,temp,ortemp[2];
    action       next_action;
    boolean      determ,continu,bound,saved;
    int          backtrack_level,global_top,trail_top,new_choice_pt,i;
    Atom         arg1_type;
    arg_list_type arg_list_1;
    arg_list_type save_args;
    mode_type    mode;
    obj_ptr      unify_add;
    global_top = global_stack->top; /* important for clean up */
    trail_top  = trail_stack->top;
    continu = true;
    backtrack_level = 0;
    saved = false;

```

```

/* Determine the argument type of the first incoming argument. */
arg1_type = (Atom)(*( dereference( arg_list, &bound ) ));
while (continuu==true) {
    switch (backtrack_level) {
        case 0:
            if ( arg1_type == atom_arr[(int)integer] ) goto shallowfail;
            else if ( arg1_type == atom_arr[(int)real] ) goto shallowfail;
                else if ( arg1_type == atom_arr[(int)str] ) determ = true;
                    else if ( arg1_type == atom_arr[(int)quoted] )
                        goto shallowfail;
                        else if ( arg1_type == atom_arr[(int)ref] )
                            determ = false;
                                else goto shallowfail;

            if ( determ == true )
                new_choice_pt = choice_point;
            else new_choice_pt = global_stack->top;
            GET_STR(nil_0,&arg_list[2*0],global_stack,
                trail_stack,new_choice_pt );
            GET_VAL(&arg_list[2*1],&arg_list[2*2],
                global_stack,trail_stack,new_choice_pt );
            mode = write;
            my_cont = NIL;
            if ( determ == true )
                cur_cont = my_cont;
            else cur_cont = continuation;
            next_action = manage_action(cur_cont,global_stack,
                trail_stack,new_choice_pt);
            delete_cont(&my_cont, continuation);
            break;
        case 1:
            if ( arg1_type == atom_arr[(int)integer] ) goto shallowfail;
            else if ( arg1_type == atom_arr[(int)real] ) goto shallowfail;
                else if ( arg1_type == atom_arr[(int)str] ) goto shallowfail;
                    else if ( arg1_type == atom_arr[(int)quoted] )
                        goto shallowfail;
                        else if ( arg1_type == atom_arr[(int)ref] )
                            determ = false;
                                else if (arg1_type == atom_arr[(int)list_2])

```



```

        determ = true;
        else goto shallowfail;
determ = true; /* last clause is always deterministic */
new_choice_pt = choice_point;
GET_STRUC(list_2,&arg_list[2*0],&mode,&unify_add,
    global_stack, trail_stack, new_choice_pt);
UNIFY_VARIABLE(&arg_list[2*3],&mode,&unify_add, global_stack);
UNIFY_VARIABLE(&arg_list[2*4],&mode,&unify_add, global_stack);
GET_STRUC(list_2,&arg_list[2*2],&mode,&unify_add,
    global_stack, trail_stack, new_choice_pt);
UNIFY_VALUE(&arg_list[2*3],&mode,&unify_add, global_stack,
    trail_stack, new_choice_pt);
UNIFY_VARIABLE(&arg_list[2*5],&mode,&unify_add, global_stack);
mode = write;
put_value( &arg_list[2*4] , &arg_list_1[2*0] );
put_value( &arg_list[2*1] , &arg_list_1[2*1] );
put_value( &arg_list[2*5] , &arg_list_1[2*2] );
if ( saved == false ) {
    saved = true;
    for (i=0; i< arglistlen; i++)
        save_args[i] = arg_list[i];
}
for (i=0; i< arglistlen; i++)
    arg_list[i] = arg_list_1[i];
arg1_type = (Atom)(*(dereference( arg_list, &bound ) ) );
goto tail_recursive;
break;
default:
    next_action = fail;
    continu = false;
    break;
}
switch ( next_action ) {
    case fail:
        if ( determ == true ) {
            continu = false;
            if ( saved == true )
                for (i=0; i<arglistlen; i++)

```

```

        arg_list[i] = save_args[i];
    }
    else {
        clean_up(global_stack,global_top,trail_stack,trail_top);
        ++backtrack_level;
    }
    break;
case determinate:
case success:
    continu = false;
    if ( determ == true ) next_action = determinate;
    break;
}
goto loop;
/* GOTO labels */
failed: if ( determ == true ) {
    next_action = fail;
    continu = false;
    if ( saved == true )
        for (i=0; i<arglistlen; i++)
            arg_list[i] = save_args[i];
}
else {
    clean_up(global_stack,global_top,trail_stack,trail_top);
    ++backtrack_level;
}
goto loop;
shallowfail: /* no stack clean_up is required */
    next_action = fail;
    ++backtrack_level;
    goto loop;
tail_recursive:
    global_top = global_stack->top;
    trail_top = trail_stack->top;
    backtrack_level = 0;
    goto loop;
loop: ;
}

```

```

    return(next_action);
}

```

Hand-optimized version of append/3

```

action append3( arg_list, continuation, global_stack,
                trail_stack, choice_point )
arg_list_type   arg_list;
contin          continuation;
g_stack        global_stack;
t_stack        trail_stack;
int            choice_point;
{
    contin      my_cont, cur_cont;
    action      next_action;
    boolean     determ, continu, bound, saved;
    int         backtrack_level, new_choice_pt, i;
    obj_ptr     *arg1;
    arg_list_type arg_list_1;
    arg_list_type save_args;
    mode_type    mode;
    obj_ptr      unify_add;
    continu = true;
    backtrack_level = 0;
    saved = false;
    while ( continu == true ) {
        arg1 = dereference( &arg_list[0], &bound ) ;
        if ( *arg1 == (obj_ptr)atom_arr[(int)str] &&
              *(arg1+1) == (obj_ptr)atom_arr[(int)nil_0] ) {
            /* CASE 0 */
            GET_VAL( &arg_list[2*1], &arg_list[2*2],
                    global_stack, trail_stack, new_choice_pt );
            continu = false;
        }
        else if ( *arg1 == (obj_ptr)atom_arr[(int)list_2] ) {
            /* CASE 1 */
            GET_STRUC( list_2, &arg_list[2*0], &mode, &unify_add,
                    global_stack, trail_stack, new_choice_pt );

```

```

UNIFY_VARIABLE( &arg_list[2*3], &mode, &unify_add, global_stack );
UNIFY_VARIABLE( &arg_list[2*4], &mode, &unify_add, global_stack );
GET_STRUC( list_2, &arg_list[2*2], &mode, &unify_add,
           global_stack, trail_stack, new_choice_pt );
UNIFY_VALUE( &arg_list[2*3],
             &mode, &unify_add, global_stack,
             trail_stack, new_choice_pt );
UNIFY_VARIABLE( &arg_list[2*5], &mode, &unify_add, global_stack );
mode = write;
put_value( &arg_list[2*4] , &arg_list_1[2*0] );
put_value( &arg_list[2*1] , &arg_list_1[2*1] );
put_value( &arg_list[2*5] , &arg_list_1[2*2] );
for (i=0; i< arglistlen; i++)
    arg_list[i] = arg_list_1[i];
}
else return(fail);
}
return(determinate);
failed;;
}

```

E.2.2 Moving Addition in Line of for/3

The first code segment shown below is the original translator output for the third clause of `for/3`. The optimization performed on this clause was to move the call to `is/2` so that it is made directly from the this predicate.

PICG Produced for/3

```

action for3(arg_list,continuation,global_stack,trail_stack,choice_point)

```

```

case 2:
    determ = true; /* last alternative */
    new_choice_pt = global_stack->top;
    GET_VAR( &arg_list[2*3], &arg_list[2*0] );
    GET_VAR( Y0, &arg_list[2*1] );

```

```

GET_VAR( Y1, &arg_list[2*2] );
mode = write;
put_unsafe( Y2, &arg_list_1[2*0], global_stack );
put_struc( plus_2, &arg_list_1[2*1], global_stack );
UNIFY_VALUE( &arg_list[2*3],
             &mode, &unify_add, global_stack,
             trail_stack, new_choice_pt );
UNIFY_INT( 1, &mode, &unify_add, global_stack,
           trail_stack, new_choice_pt );
put_value( Y2, &arg_list_2[2*0] );
put_value( Y0, &arg_list_2[2*1] );
put_value( Y1, &arg_list_2[2*2] );
my_cont = NIL;
new_contin(&temp);
temp->type = and;
temp->elem.func.ptr = for3;
temp->elem.func.args = arg_list_2;
temp->elem.func.next = my_cont;
my_cont = temp;
new_contin(&temp);
temp->type = and;
temp->elem.func.ptr = is2;
temp->elem.func.args = arg_list_1;
temp->elem.func.next = my_cont;
my_cont = temp;
cur_cont = add_rem( my_cont, continuation );
next_action = manage_action(cur_cont, global_stack,
                           trail_stack, new_choice_pt );
delete_cont(&my_cont, continuation);
break;

```

```

}

```

Optimized Third Clause of for/3

```

action for3(arg_list, continuation, global_stack, trail_stack, choice_point)

```

```

:
case 2:
    determ = true; /* last alternative */
    new_choice_pt = choice_point;
    GET_VAR( &arg_list[2*3], &arg_list[2*0] );
    GET_VAR( Y0, &arg_list[2*1] );
    GET_VAR( Y1, &arg_list[2*2] );
    mode = write;
    put_unsafe( Y2, &arg_list_1[2*0].global_stack );
    iters = dereference( &arg_list[0], &bound );
    index = dereference( Y2, &bound );
    *(index) = (obj_ptr)atom_arr[(int)integer];
    *(index+1) = (obj_ptr)malloc(sizeof(integer));
    *(int *)*(index+1) = *(int *)*(iters+1) + 1;
    put_value( Y2, &arg_list_2[2*0] );
    put_value( Y0, &arg_list_2[2*1] );
    put_value( Y1, &arg_list_2[2*2] );
    my_cont = NIL;
    new_contin(&temp);
    temp->type = and;
    temp->elem.func.ptr = for3;
    temp->elem.func.args = arg_list_2;
    temp->elem.func.next = my_cont;
    my_cont = temp;
    cur_cont = add_rem( my_cont, continuation );
    next_action = manage_action(cur_cont,global_stack,
                                trail_stack,new_choice_pt );
    delete_cont(&my_cont, continuation);
    break;

```

E.3 Exam Scheduler

The test program is as follows:

```
module scheduler.
```

```

all_global.
body.
dynamic(avail_slots/2).  /* avail_slots(prof id, available time slot list) */
                        /* stored directly from input data */

dynamic(committee/3).    /* committee(student id, faculty member list, */
                        /*           time slot) stored directly from input */
                        /* data with initial uninstantiated time slot */

dynamic(tight/2).        /* tight(student id = committee id, # of potential */
                        /*           time slots) initially set to notime slots */
                        /* this measures how tight a committee schedule is */
                        /* by the number of potential time slots */

dynamic(taken/2).        /* taken(prof id, time slot) */

dynamic(potential/2).    /* potential(student id = committee id, potential */
                        /*           time slot ) */

/******
/* Main Line
/*
/* The program writes out a header to indicate that it has begun. It queries*/
/* the user for a data filename, and processes the data.
/*
/******
schedule_PhD_orals :-
    write("***** PhD Oral Exam Scheduler *****"),nl(3),
    write("Enter information filename"),nl,
    read(X),
    read_token(expression end),
    set_channel(infile,X),
    set_input,
    read,
    read_input_line,
    close_input, write_committee,
    potential_meetings, write_pots,
    ( not(tight(_,0)),          /* tight(_,0) implies atleast one */

```

```

                                /* committee could not be convened. */
    schedule_profs, write_taken, write_committee,
    write_successful_solution ;
    write_failed_solution ).

/******
/*
/*      Input Reader
/*
/*
/*  The read_input_line/0 predicate will read the contents of a correctly
/*  formatted data file.  Information entered more than once for the same
/*  professor or student is replaced with the last entered set of data.
/*
/******
read_input_line :- read_token(file end).
read_input_line :- read(X), read_token(expression end), cut_input,
                    process_line(X).

process_line(p-PID-AVAIL) :- ( del_statement(avail_slots(PID,_)),';
                             succeed ),
                             add_statement(avail_slots(PID,AVAIL)),
                             fail.

process_line(s-SID-C)      :- ( del_statement(committee(SID,_,_)),';
                             succeed ),
                             add_statement(committee(SID,C,SLOT)),
                             ( del_statement(tight(SID,0));
                             succeed ),
                             add_statement(tight(SID,0)),
                             /* initialize tight/2 to simplify the test */
                             /* for possible conventions */
                             fail.

/******
/*  Testing for Possible Convening
/*
/*
/*  This section will determine if the committees can be individually
/*  convened.  It will keep track of potential time slots for a given
/*  committee, and how many potential time slots exist is stored in the
/*  tight/2 fact.  These facts are stored in ascending order so that,
/*  at the end of this processing, the committee with the tightest schedule

```



```

/* will be at the top of the clause list. */
/* */
/* potential_meetings/0 is a failure driven loop which tries all slots for */
/* all committees, where all slots is reduced to the slots of one of the */
/* professors in the committee. The member/2 predicate is used to generate */
/* a time slot from the available slots of one of the professors, and */
/* when all of these time slots are exhausted the program will backtrack */
/* into the next committee. When all committees have been examined this */
/* will fail. */
/* */
/* If all faculty members are available in a given time slot then the */
/* tightness count is incremented. The more time slots which fit a */
/* committee, the higher this number, and the easier it should be to schedule */
/* this committee. */
/* */
/*/*****
potential_meetings :- committee(SID,[H|T],SLOT),
                        avail_slots(H,AVAIL),
                        member(Y,AVAIL),
                        determine_potential_slots(SID,Y,T).

potential_meetings.

determine_potential_slots(SID,SLOT,C) :-
    all_avail_in_slot(SLOT,C), /* failing here moves */
    /* the examination to the next time slot or committee */
    add_statement(potential(SID,SLOT)),
    del_statement(tight(SID,NUM)),
    NUM2 is NUM + 1,
    add_statement(tight(SID,NUM2)),
    fail. /* fail to drive the potential meetings loop */

all_avail_in_slot(SLOT,[H|T]) :- avail_slots(H,AVAIL),
                                member(SLOT,AVAIL), !,
                                all_avail_in_slot(SLOT,T).

all_avail_in_slot(_,[]).

/*/*****

```

```

%/* Scheduling the Committees */
%/* */
%/* If all committees are individually convenable, then an attempt to form a */
%/* joint schedule is made. The scheduling starts with the tightest committee*/
%/* to schedule. The scheduling completes when all committees are scheduled. */
%/* This is equivalent to all tight/2 facts being removed. Committees are */
%/* slotted from their potential slot facts. Once a slot is chosen, a */
%/* taken/2 fact is recorded for each professor in that time slot. When a */
%/* conflict arises, a new potential time slot is chosen in the most recently */
%/* scheduled committee which can be re-scheduled according to the taken/2 */
%/* restrictions. Schedule_profs/0 will fail if a schedule cannot be found. */
%/* */
%/* *****/
schedule_profs :- del_statement_b(tight(SID,_)),
                  schedule_committee(SID),
                  schedule_profs.
schedule_profs :- not(tight(_,_)). %/* succeeds if all tight/2 facts have */
                                %/* been removed => all committees are scheduled */

schedule_committee(SID) :-
    del_statement_b(committee(SID,C,SLOT)),
    slot_committee(SID,C,SLOT).

slot_committee(SID,C,SLOT) :-
    potential(SID,SLOT), write,SLOT), nl,
    slot_profs(SID,C,SLOT), write,SLOT), nl,
    add_statement_b(committee(SID,C,SLOT)),
    write_committee.

slot_profs(SID,[PROF|T],SLOT) :-
    not(taken(PROF,SLOT)),!,
    add_statement_b(taken(PROF,SLOT)),
    slot_profs(SID,T,SLOT).
slot_profs(_,[],_).

%/* *****/
%/* Output predicates */
%/* *****/

```

```

write_successful_solution :- nl(3),
    write("      *** PhD Oral Exam Schedule ***"),nl(2),
    not(write_each_committee),nl(3).
write_each_committee :- committee(SID,C,SLOT),
    write("      committee = "),
    write(SID),write_tab(29),
    write(" - "),write(C),write_tab(52),
    write(" at "),write,SLOT),nl,
    fail.
write_pots :- potential(SID,SLOT),
    write(SID),write(" - "),write,SLOT),nl,fail.
write_pots.
write_taken :- taken(P,SLOT),
    write(P),write(" - "),write,SLOT),nl,fail.
write_taken.
write_committee :- committee(S,P,SLOT),
    write(S),write(" - "),
    write(P),write(" - "),write,SLOT),nl,fail.
write_committee.
write_failed_solution :- nl(3),
    write("The exams cannot be scheduled with the data given."),nl(3).

%/*****/
member(X,[X|Xs]).
member(X,[Y|Ys]) :- member(X,Ys).
endmod.

```

E.3.1 Optimization of all_avail_in_slot/2

The result of optimizing all_avail_in_slot/2 by making determinate calls directly and simplifying the tail recursive nature to a loop is shown below. The original output is not given.

```

action all_avail_in_slot2(arg_list,continuation,global_stack,
    trail_stack,choice_point)
arg_list_type    arg_list;

```

```

contin      continuation;
g_stack     global_stack;
t_stack     trail_stack;
int         choice_point;
{
    contin      my_cont, cur_cont, temp, ortemp[2];
    action      next_action;
    boolean     determ, continu, bound, saved;
    int         backtrack_level, global_top, trail_top, new_choice_pt, i;
    Atom        arg2_type;
    arg_list_type arg_list_1, arg_list_2, arg_list_3, arg_list_4;
    arg_list_type save_args;
    arg_list_elem Y0, Y1, Y2;
    mode_type    mode;
    obj_ptr      unify_add;
    global_top = global_stack->top; /* important for clean up */
    trail_top  = trail_stack->top;
    continu = true;
    backtrack_level = 0;
    saved = false;
    /* Determine the argument type of the SECOND incoming argument. */
    arg2_type = (Atom)(*(dereference( &arg_list[2], &bound )));
    while (continu==true) {
        if ( arg2_type == atom_arr[(int)list_2] ) {
            determ = false; /* first arg_type switch not necessary */
            new_choice_pt = global_stack->top;
            GET_VAR( Y0, &arg_list[2*0] );
            GET_STRUC( list_2, &arg_list[2*1], &mode, &unify_add,
                global_stack, trail_stack, new_choice_pt );
            UNIFY_VARIABLE( &arg_list[2*2], &mode, &unify_add, global_stack );
            UNIFY_VARIABLE( Y1, &mode, &unify_add, global_stack );
            mode = write;
            put_struc( avail_slots_2, &arg_list_1[2*0], global_stack );
            UNIFY_VALUE( &arg_list[2*2],
                &mode, &unify_add, global_stack,
                trail_stack, new_choice_pt );
            UNIFY_VARIABLE( Y2, &mode, &unify_add, global_stack );
            put_value( Y0, &arg_list_2[2*0] );

```

```

    put_value( Y2, &arg_list_2[2*1] );
    put_value( Y0, &arg_list_4[2*0] );
    put_value( Y1, &arg_list_4[2*1] );
    if ( ( next_action = spy_dyn1( arg_list_1, NIL, global_stack,
                                trail_stack, new_choice_pt ))
        == fail ) goto failed;
    if ( (next_action = member2( arg_list_2, NIL, global_stack,
                                trail_stack, new_choice_pt ))
        == fail ) goto failed;
    for (i=0; i< arglistlen; i++)
        arg_list[i] = arg_list_4[i];
    arg2_type = (Atom)(*(dereference( &arg_list[2], &bound ) ) );
}
else {
    continu = false;
    if ( arg2_type == atom_arr[(int)str] )
        next_action = determinate;
    else next_action = fail;
}
}
return(next_action);

/* GOTO labels */
failed: /* clean-up global stack */
    next_action = fail;
    clean_up(global_stack,global_top,trail_stack,trail_top);
return(next_action);
}

```

E.4 Interpreter Test

The LISP interpreter test program is given below:

```

module executer.
/******
/*
/*          LISP INTERPRETER
/*

```

```

%/*                                                                    */
%/* Program Description:                                              */
%/*   This program implements the basic components of a LISP interpreter, */
%/*   namely:                                                         */
%/*     (a) Lisp reader                                              */
%/*     (b) Lisp writer                                              */
%/*     (c) Lisp executor                                            */
%/*                                                                    */
%/* The language syntax being used is:                                */
%/*   <expr> :- <quoted_expr> | <basic_expr>                          */
%/*   <quoted_expr> :- * <basic_expr>                                */
%/*   <basic_expr> :- <atom> | <list>                                  */
%/*   <list> :- <actual_list> | <empty_list>                          */
%/*   <empty_list> :- () | nil                                       */
%/*   <actual_list> :- ( <expr> {<space> <expr>} )                  */
%/*   <atom> :- <mprolog_string> | <mprolog_integer>                  */
%/*                                                                    */
%/* The program will prompt the user for a legal <expr>, which is read */
%/* by the Lisp Reader, executed by the Lisp Executor to produce a   */
%/* function value which is, in turn, printed by the Lisp Printer.   */
%/*                                                                    */
%/* The program is terminated by entering (exit) to the lisp prompt. */
%/*                                                                    */
%/* The program supports the following built-in functions:          */
%/*   car, cdr, setq, + (dyadic), - (dyadic), quote, cons, reverse, */
%/*   last, append, length, and exit.                                */
%/*                                                                    */
%/* THE PIC ENVIRONMENT WAS UNABLE TO MANAGE THE ORIGINAL LISP READER. */
%/* THE USER MUST THEREFORE PERFORM THE READER OPERATIONS MANUALLY   */
%/* *****/
all_global.
body.
dynamic(atom/2).
initialize_environment :- add_statement(atom(nil,[])).
%/* *****/
%/*   LISP EXECUTOR                                                  */
%/*                                                                    */
%/* The Lisp Executor continually reads a Lisp expression from the  */

```

```

/* terminal, evaluates the expression and prints the solution. The */
/* process is terminated by the expression (exit). */
/* */
/* The executor recursively determines the values of the expression */
/* arguments before determining the value of the expression. */
/******
lisp :- initialize_environment,                /* INITIALIZATION */
        repeat,
        read(X), read_token(expression end), cut_input,
        (X=[exit],write("{return to PDSS}"));
        execute(X,SOL),print_solution(SOL),nl,fail.

execute([],[]) :- !.
execute([quote|T],T) :- !.
execute([setq,SYM,EXPR],SOL) :- !,
        execute(EXPR,SOL),
        solve([setq,SYM,SOL],SOL).
execute([FUNCTOR|ARGS],SOL) :-
        built_in(FUNCTOR),!,
        execute_args(ARGS,NEWARGS),
        solve([FUNCTOR|NEWARGS],SOL).
execute(ATOM,ATOM) :-
        ATOM is_a number,!.
execute(ATOM,VALUE) :-
        ATOM is_a string, !,
        (atom(ATOM,VALUE);
        signal("undefined_variable")).
execute(.,_) :- signal("undefined_function").

execute_args([ARG1|T],[NARG1|NT]) :- !,execute(ARG1,NARG1),
        execute_args(T,NT).

execute_args([],[]).
/******
/*      SOLVE SIMPLIFIED LISP EXPRESSION */
/* */
/* Module Solve determines the value of a simplified Lisp expression. */
/* It evaluates built-in functions. */
/******

```

```

solve([car,[H|T]],H) :- !.
solve([cdr,[H|T]],T) :- !.
solve([" + ",A,B],SOL) :- SOL is A+B,!
solve([" - ",A,B],SOL) :- SOL is A-B,!
solve([cons,X,[H|T]],[X,H|T]) :- !.
solve([cons,X,[]],[X]) :- !.
solve([reverse,[H|T]],SOL) :- reverse([H|T],[],SOL),!.
solve([last,[H|T]],SOL) :- last([H|T],SOL),!.
solve([append|T],SOL) :- append(T,SOL),!.
solve([length,[H|T]],SOL) :- list_len([H|T],SOL),!.
solve([setq,SYM,VAL],VAL) :- (atom(SYM,_),
    del_statement(atom(SYM,_));succeed),
    add_statement(atom(SYM,VAL)),!.
solve(,_):- signal("misused_function").
%/******
%/*          BUILT-IN FUNCTORS DATA BASE AND BUILT-IN FUNCTIONS          */
%/*          */
%/* This data base contains the symbols which represent regular,          */
%/* non-special, built-in functions.                                     */
%/* This module also contains the built_in predicate functions.          */
%/******
built_in(car).
built_in(cdr).
built_in(cons).
built_in(" + ").
built_in(" - ").
built_in(append).
built_in(reverse).
built_in(last).
built_in(length).
built_in(quote).

reverse([X|Xs],ACC,Ys) :- reverse(Xs,[X|ACC],Ys).
reverse([],Ys,Ys).

last([X|Xs],ANS) :- last(Xs,ANS).
last([X],[X]).

```



```

append([L1,L2],SOL) :- append(L1,L2,SOL).
append([L1,L2|T],SOL) :- append(L1,L2,TEMP),
    append([TEMP|T],SOL).
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).

list_len([],0).
list_len([H|T],NUM) :- list_len(T,N), NUM is N + 1.

%*****/
%/*          LISP PRINTER                                     */
%/*          */
%/*  The Lisp Printer writes the lisp expression's solution to the terminal */
%/*  in lisp notation.  That is, the Prolog lists are written as lisp lists */
%/*  with parentheses as opposed to square brackets.                        */
%*****/
print_solution([H|T]) :- !,write(" ( "),
    print_solution(H),
    print_tail(T),
    write(" ) ").
print_solution([]):- !,write(" () ").
print_solution(X) :- !,write(X).

print_tail([H|T]) :- write(" "),
    print_solution(H),
    print_tail(T).
print_tail([]).

%*****/
%/*          LISP INTERPRETER EXCEPTION HANDLER               */
%*****/
signal(X) :- print_message(X), fail.

print_message(undefined_variable) :- !, nl,
    write("Undefined Variable"),nl.
print_message(undefined_function) :- !, nl,
    write("Undefined Function"),nl.
print_message(misused_function) :- !, nl,
    write("Misused Function"),nl.

```

```
print_message(X) :- nl,  
    write("Undetermined Error"),nl.  
endmod.
```

E.4.1 Optimized Version of built_in/1

The optimized version of `built_in/1` is given in Figure E.1.

```

action built_in1(arg_list,continuation,global_stack,trail_stack,choice_point)

:

obj_ptr      *arg1;

:

arg1 = dereference( arg_list, #bound ) ;
if ( *arg1 == (obj_ptr)atom_arr[(int)str] ) {
    if ( *(arg1 + 1) == (obj_ptr)atom_arr[(int)car_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)cdr_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)cons_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)append_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)reverse_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)last_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)length_0] ||
        *(arg1 + 1) == (obj_ptr)atom_arr[(int)quote_0] )
        return(determinate);
    else return(fail);
}
else
    if ( *arg1 == (obj_ptr)atom_arr[(int)quoted] ) {
        if ( strcmp( (arg1 + 1), " + ") ||
            strcmp( (arg1 + 1), " - ") )
            return(determinate);
        else return(fail);
    }
    else return(fail);
}

```

Figure E.1: Optimized version of built_in/1.

```

main()
{
    g_stack          global_stack;
    t_stack          trail_stack;
    arg_list_type     arg_list;
    action            next_action;
    arg_list_elem     Y1, Y2;

    global_stack = (g_stack)malloc(sizeof(g_stack_type));
    read_stack = (r_stack)malloc(sizeof(r_stack_type));
    trail_stack = (t_stack)malloc(sizeof(t_stack_type));

    init_atomtable();
    global_infile_name = (char *)malloc(16);
    global_outfile_name = (char*)malloc(16);

    next_action = nrev0(arg_list, NIL, global_stack,
                        trail_stack, 0 );
}

```

Figure E.2: Minimum mainline and CPI requirements for `nrev/0`.

END

1 8·0 2·9 0

FIN