# ECTOPIC — An Extended Translator of Prolog into C++

by

Carlos Escalante
Licentiateship (B.Sc.), Universidad Iberoamericana, Mexico, 1988

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

ACCEPTED
CULTY OF GRADUATE STUDIES
DEAN
E Sept. 28/92

We accept this thesis as conforming
to the required standard

Dr. R. N. Horspool, Supervisor (Department of Computer Science)

Dr. H.A. Müller, Departmental Member (Department of Computer Science)

Dr. P. Agathoklis, Outside Member (Department of Electrical Engineering)

Dr. Inna Sharf, External Examiner (Department of Mechanical Engineering)

© Carlos Escalante, 1992

University of Victoria

All rights reserved. Thesis may not be reproduced in whole or in part, by
photocopy or other means, without the permission of the author.

Supervisor: Dr. R. Nigel Horspool.

## ABSTRACT

This thesis describes the ECTOPIC system, a translator of Prolog to C++. The translator, entirely programmed in Prolog, includes two important optimizations: tail recursion optimization and clause indexing. ECTOPIC is an enhancement of the TOPIC system, a University of Victoria-IBM project, which supports multi-lingual or mixed paradigm programming, but without incorporating some crucial optimizations such as tail recursion optimization and clause indexing. To implement these optimizations, a static analysis is performed to determine which Prolog predicates are determinate, using a method due to Debray. With the incorporation of these optimizations, ECTOPIC has been able to produce much better code than TOPIC for certain programs.

Examiners:

Dr. R. N. Horspool, Supervisor (Department of Computer Science)

Dr. H.A. Müller, Departmental Member (Department of Computer Science)

Dr. P. Agathoklis, Outside Member (Department of Electrical Engineering)

Dr. Inna Sharf, External Examiner (Department of Mechanical Engineering)

# Table of Contents

Abstract . . . . . . . . . . ii

Table of Contents . . . . . . . . . . iii

List of Tables . . . . . . . . . . v

List of Figures . . . . . . . . . . vi

Acknowledgments . . . . . . . . . . vii

**Chapter 1 Introduction** **1**

**Chapter 2 Two Programming Paradigms** **3**

2.1 The Object-Oriented Programming Paradigm . . . . . . . . . . 3

2.1.1 Data Abstraction and Information Hiding . . . . . . . . . . 3

2.1.2 Inheritance . . . . . . . . . . 4

2.1.3 Polymorphism . . . . . . . . . . 5

2.2 C++ as an Example an Object-Oriented Language . . . . . . . . . . 6

2.2.1 The Class Concept . . . . . . . . . . 6

2.2.2 Inheritance in C++ . . . . . . . . . . 7

2.2.3 Operator Overloading . . . . . . . . . . 8

2.2.4 Polymorphism in C++ . . . . . . . . . . 9

2.3 Prolog and Logic Programming . . . . . . . . . . 9

2.3.1 Prolog, a Logic Programming Language . . . . . . . . . . 10

2.3.2 Prolog Terms . . . . . . . . . . 10

2.3.3 Prolog Clauses . . . . . . . . . . 11

2.3.4 Searching Strategy in Prolog; Unification and Backtracking . . . . . 12

2.3.5 Extra-logical Features in Prolog . . . . . . . . . . 13

**Chapter 3 An Overview of The Warren Abstract Machine** **15**

3.1 The Warren Abstract Machine (WAM) . . . . . . . . . . 15

3.2 WAM Model Calling Mechanisms . . . . . . . . . . 16

3.3 Term Representation . . . . . . . . . . 17

3.4 Clause Indexing . . . . . . . . . . . . . . . . . . . . . . . . . . . . .21

**Chapter 4 The TOPIC System** **25**

4.1 Introduction to the Topic System . . . . . . . . . . . . . . . . . . . . . .25

4.2 The TOPIC Translation Scheme . . . . . . . . . . . . . . . . . . . . . .26

4.2.1 Control Flow . . . . . . . . . . . . . . . . . . . . . . . . . .26

4.2.2 Memory Organization. . . . . . . . . . . . . . . . . . . . . . . .28

4.3 Prolog Objects Representation . . . . . . . . . . . . . . . . . . . . . .33

4.4 Unification. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .36

4.5 Backtracking and Frame Allocation. . . . . . . . . . . . . . . . . . . .36

4.6 Cut Operation . . . . . . . . . . . . . . . . . . . . . . . . . . . . .37

**Chapter 5 Mode analysis** **40**

5.1 Modes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .40

5.2 General Mode Analysis Method . . . . . . . . . . . . . . . . . . . . .41

**Chapter 6 ECTOPIC, an Improved Translator** **47**

6.1 A Translator Written in Prolog . . . . . . . . . . . . . . . . . . . . .47

6.2 Adding Prolog Optimizations to the Translator. . . . . . . . . . . . . . .48

6.2.1 Mode Analysis . . . . . . . . . . . . . . . . . . . . . . . . .48

6.2.2 Determinacy Analysis. . . . . . . . . . . . . . . . . . . . . . .49

6.2.3 Tail Recursion Optimization . . . . . . . . . . . . . . . . . . . .50

6.2.4 Clause Indexing . . . . . . . . . . . . . . . . . . . . . . . . .51

**Chapter 7 Experimental Results and Conclusions** **55**

7.1 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .55

7.2 Conclusions and Future Work. . . . . . . . . . . . . . . . . . . . . . .57

**Bibliography** **60**

**Appendix 1 Test Programs** **64**

# List of Tables

**Table 3.1.** Register assignment for the examples in Figures 3.1 and 3.2. . . . . . 19
**Table 3.2.** Different possibilities for the unification of two terms . . . . . . . . . . 24
**Table 4.1.** Term creation in the TOPIC system . . . . . . . . . . . . . . . . . . . 34
**Table 4.2.** Declaration of queries in the TOPIC system . . . . . . . . . . . . . . . 35
**Table 4.3.** Definition of heads and rules in the TOPIC system . . . . . . . . . . . 35
**Table 7.1.** TRO benchmark. . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
**Table 7.2.** Path finder program benchmark . . . . . . . . . . . . . . . . . . . . . 58
**Table 7.3.** Query example benchmark . . . . . . . . . . . . . . . . . . . . . . . 58

# List of Figures

**Figure 2.1.** A typical class hierarchy from Smalltalk . . . . . . . . . . . . . . . . 5
**Figure 2.2.** Example of polymorphism in Smalltalk . . . . . . . . . . . . . . . . 5
**Figure 3.1.** Partial translation of a clause head . . . . . . . . . . . . . . . . . . . . 19
**Figure 3.2.** Partial translation of a query . . . . . . . . . . . . . . . . . . . . . 20
**Figure 3.3.** Environment trimming . . . . . . . . . . . . . . . . . . . . . . . 22
**Figure 3.4.** An example of clause indexing . . . . . . . . . . . . . . . . . . . . 23
**Figure 3.5.** The general translation scheme for predicates . . . . . . . . . . . . 23
**Figure 4.1.** The box representation of a subgoal . . . . . . . . . . . . . . . . . 27
**Figure 4.2.** The box representation of a goal . . . . . . . . . . . . . . . . . . . 27
**Figure 4.3.** The box representation of a predicate . . . . . . . . . . . . . . . . . 29
**Figure 4.4.** Translation scheme to reflect the flow control of a predicate . . . . 30
**Figure 4.5.** Box representation for a predicate . . . . . . . . . . . . . . . . . . 31
**Figure 4.6.** TOPIC's state registers . . . . . . . . . . . . . . . . . . . . . . . 32
**Figure 4.7.** The TOPIC class hierarchy for Prolog terms . . . . . . . . . . . . . 33
**Figure 4.8.** Use of the Unify and Connect functions . . . . . . . . . . . . . . . 37
**Figure 4.9.** Backtracking management for a typical Prolog predicate . . . . . . 38
**Figure 4.10.** The box representation of the cut predicate . . . . . . . . . . . . . 39
**Figure 5.1.** Abstract interpretation applied to Prolog unification . . . . . . . . . 42
**Figure 5.2.** Instantiation and sharing information in a goal . . . . . . . . . . . . 43
**Figure 5.3.** Instantiation patterns for a particular call . . . . . . . . . . . . . . . 44
**Figure 5.4.** Inherited instantiations for some term instantiations . . . . . . . . . 45
**Figure 5.5.** Variable instantiations after unification . . . . . . . . . . . . . . . 45
**Figure 5.6.** Aliasing information of variables after unification . . . . . . . . . . 46
**Figure 6.1.** Mode analysis can improve the results of determinacy analysis . . . 50
**Figure 6.2.** Tail recursion optimization in the TOPIC system . . . . . . . . . . 52
**Figure 6.3.** Control flow for clause indexing . . . . . . . . . . . . . . . . . . . 53
**Figure 7.1.** The naïve reverse program . . . . . . . . . . . . . . . . . . . . . 55
**Figure 7.2.** The quicksort example . . . . . . . . . . . . . . . . . . . . . . . 57

## ACKNOWLEDGEMENTS

I would like to thank Dr. Horspool for his patience and encouragement; Dr. Müller for his interest and advice during the writing stages; James Uhl, who offered a number of insights; and, finally, last but certainly not least, my parents and brother, for their long-standing devotion and support.

# Chapter 1. Introduction

This thesis shows how the code generated by the TOPIC system [JLH90] can be substantially improved when standard Prolog optimization techniques are applied. The code generated by ECTOPIC, our enhanced translator, has proven to be more efficient for many programs via tail recursion optimization and clause indexing techniques.

Traditionally, application programs are developed using one programming language exclusively. However it is often the case that, for a given specific application, no programming language proves to be an absolute best choice. It is not surprising that several authors have pointed out that a trend in program development will increasingly be towards *multi-language programming* [Hai86] [HMS88] [Hug91] [Wil91], that is, the adoption of solutions that are written in more than just one programming language. With this approach, a programming language can be advantageously used in those parts of an application for which it is well-suited. Thus, pieces of code written in different programming languages can be amalgamated to build a complex application.

One programming tool that attempts to support this *mixed* paradigm is the TOPIC system [JLH90], which permits the development of programs in both the C++ and Prolog languages. It therefore allows the simultaneous use of three of the most important programming *paradigms* or models for programming: logic programming, imperative and object-oriented programming. Logic programming [Kow79], supported by Prolog, is oriented to those applications that are concerned with symbolic logic and reasoning, such as artificial intelligence and expert systems. Object-oriented programming [Weg86] is a paradigm supported by the C++ programming language. It is essentially based on the definition of objects that have a state and associated operations, allowing the inheritance of properties. C++ is considered to be a general-purpose programming language in the sense that it is intended to cope with every major application area.

The core of the TOPIC system is a translator from Prolog into C++. Since C++ can be converted into C code, the resulting code usually achieves the advantages of portability and code close in efficiency to the resulting assembly language. However, the TOPIC system does not generate efficient C++ code. It is the main purpose of this thesis to show how the TOPIC system can produce better code by means of applying some standard Prolog optimization techniques, in particular *tail recursion optimization* and *clause indexing*. The application of such techniques requires a static analysis of the Prolog source code to establish which predicates are deterministic (in the sense that the predicate is used to obtain at most one answer) and which predicate arguments are always instantiated to a ground term. Both *determinacy* and *mode analysis* have been extensively studied in the literature [Mel85] [DW88].

With the incorporation of tail recursion optimization and clause indexing into the TOPIC system, we have been able to generate better code than that created by the TOPIC system. Generally speaking, those programs with a tail-recursive nature are executed with better management of the stack. Similarly, by virtue of clause indexing, database-like applications usually execute faster in ECTOPIC.

The remainder of this thesis is arranged as follows: Chapter 2 presents a brief overview of Prolog and C++, the languages that are related to the TOPIC system. Chapter 3 explores some characteristics of the Warren Abstract Machine, an architecture that has become the *de facto* basis of Prolog compilers. Chapter 4 gives an overall description of the TOPIC system. Chapter 5 is devoted to the usability of mode analysis for Prolog programs, particularly a framework developed by Debray [Deb89] which takes into account the treatment of aliasing. Chapter 6 describes how determinacy analysis is performed to add tail recursion optimization and clause indexing to the TOPIC system. Chapter 7 gives some experimental results. Finally, Chapter 8 presents some conclusions and suggests some directions for further work related to this thesis.

# Chapter 2. Two Programming Paradigms

As computer science has evolved, several programming paradigms have appeared [Wat90]. Each one emphasizes a particular approach to programming. The standard paradigms include: imperative, object oriented, concurrent programming, functional programming, logic programming, and relational programming. Whenever a new programming paradigm is proposed, several new programming languages or extensions of previously existing ones are usually invented to support it.

In this section, a brief description of the two programming paradigms relevant to the TOPIC system is given, namely, the object-oriented and the logic programming paradigms.

## 2.1 The Object-Oriented Programming Paradigm

Abstraction is often proposed as a fundamental tool for solving problems. In the programming discipline, abstraction has been applied to data types, control constructs, strategies for defining modules, program organization and structure, and so on.

A milestone in the evolution of programming languages occurred with the development of the notion of an *abstract data type* [Sha84]. An abstract data type represents sets of entities or values that have some distinctive properties associated with them (generally some operations that can be applied to those values). Such entities can be regarded as abstract *objects* that represent concrete entities. It is not surprising that one programming approach that has become widely accepted is the *object-oriented* paradigm [Weg84] [Str88], which is based on the concept of an *object* as a software entity which is characterized by a specific behaviour.

The basic ideas of the object-oriented paradigm can be grouped into the following categories: mechanisms for data abstraction and information hiding, inheritance issues, and polymorphism support. All these concepts are covered in the following sub-sections.

### 2.1.1 Data Abstraction and Information Hiding

A data abstraction is a concise representation of some object, usually hiding many accidental or irrelevant details. A data abstraction usually specifies both the data structure and its interface to the exterior. In object-oriented programming languages, an instance of a data abstraction is called an *object* [Wol89]. The implicit data structure for the object consists of data fields, but with the usual restriction that users can have access to the object only through a set of functions defined within the data abstraction. Thus, the object is described by means of its data and some function definitions (usually called

*methods*). The idea of information hiding or encapsulation is basic to the object-oriented paradigm.

*Encapsulation* is a process in which individual components are defined in such a way that (1) there is a clear boundary defining their scope; (2) there is a well-defined interface that permits access to the components; (3) the programmer is encouraged to protect or hide the internal implementation.

The components of the data structure are subject to a rigid set of visibility rules that determine which elements are visible to the environment and what conditions must hold to grant access (via the interface). The object-oriented paradigm creates a separation between the definition and the implementation parts of an abstract data type. An important advantage of this approach is the possibility to modify the implementation without affecting the user's code.

### 2.1.2 Inheritance

Inheritance describes the transmission of characteristics or properties from an ancestor to a descendant. In object-oriented programming languages, the inheritance mechanisms allow extension of the data-abstraction definition facilities. In other words, the code and behaviour that are associated with a particular abstract data type can be shared or refined by new abstract data types without having to repeat everything again. For the purposes of the object-oriented programming paradigm, the ancestor abstract data type is referred to as a *base* class (or super-class) and the corresponding descendants are called the *derived* classes (or *sub-classes*).†

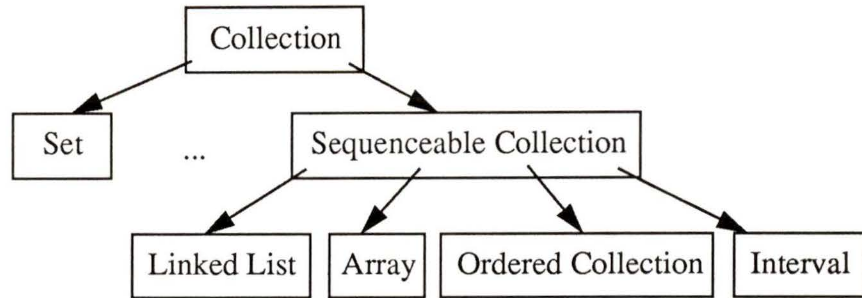
When one class is derived from a base class, the derived class normally inherits all the data fields and methods available in the base class. Naturally, the derived class can provide additional data fields and methods. An advantage that derives from this capability is that some generic (basic) objects can be refined into more specialized classes that are more suitable to a particular application, encouraging the principle of reusability.

A derived class can itself be a base class for new derived classes, thereby creating a class hierarchy (cf. Figure 2.1)‡. Furthermore, inheritance from more than one base class, that is, *multiple inheritance* is also possible (although some languages do not support this generalization).

---

† Some object-oriented languages use a different scheme for inheritance, called *delegation*, which is an object-based inheritance rather than a class-based inheritance via the utilization of prototypes [Lie86].

‡ Smalltalk-80 is a trademark of Xerox Corporation

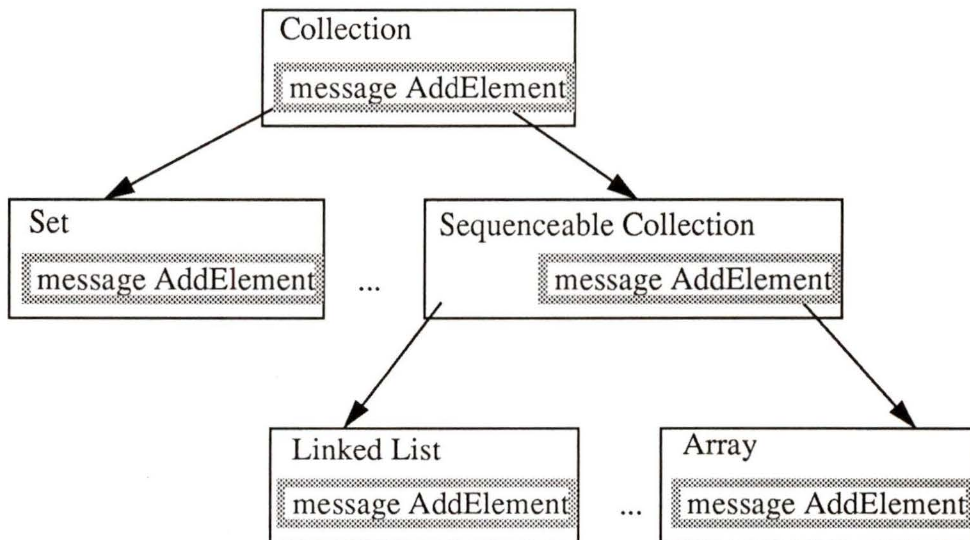



The base class *Collection* is refined through more complex derived classes.

**Figure 2.1** A typical class hierarchy from Smalltalk.

### 2.1.3 Polymorphism

*Simple polymorphism* occurs when a concrete operation inherits its definition and properties from a generic operation [HN87]. Under the object-oriented programming point of view, polymorphism permits the same *message* (that is, a request for an object to carry out one of its operations) to be sent to different kinds of objects, probably causing different actions to be performed. Thus, a generic interface defined by a base class can be applied uniformly to all the derived classes in spite of different internal representations. Usually, a derived class can define the particular method that must be applied to an instance of itself (cf. Figure 2.2).




The same message *AddElement* has a different meaning depending on the context in which it appears (for example, a *Set* object implies that element repetitions do not occur).

**Figure 2.2** Example of polymorphism in Smalltalk.

The straightforward mechanism for allowing polymorphism is via dynamic binding. This normally means that the precise method to be applied should be determined at run-time, since the instance the method is applied to is not known in advance.

Polymorphism is often implemented in terms of *abstract classes*, which are special objects that are defined only to be used as a starting point in the definition of new classes, and are not supposed to have instantiations of their own. The class *Collection* in Figure 2.2 is a typical example of an abstract class. The methods of an abstract class usually do not perform any actions, and they must be redefined by the (concrete) derived classes.

## 2.2 C++ as an Example an Object-Oriented Language

C++ [Str86] [Jor90] has become one of the most popular object-oriented programming languages, perhaps because C++ is an extension, or a superset, of the ubiquitous C programming language.

C++ fully supports all the mechanisms that are associated with the object-oriented programming paradigm: definition of objects (collections of operations that share a state), information hiding (which permits the internal states to be protected from the outside world), re-use of the behaviour of a base class in the definition of a derived class or sub-class (inheritance), etc. But, unlike other object-oriented languages (Smalltalk, for instance), C++ does not permit dynamic modifications to class definitions.

Object-oriented programming creates a setting in which dynamic objects interact by sending messages to each other. In C++, such messages correspond to procedure calls. Some of the abstraction mechanisms that support the object-oriented paradigm in C++ are studied in more detail within the following sub-sections.

### 2.2.1 The Class Concept

C++ adopted and extended the class concept from a programming language that is regarded as a pioneer in some respects: the SIMULA-67 language [BDMN73]. Classes in C++ are closely related to the traditional concept of data type. *Classes* are abstract data types, and *objects* are instances of these classes.

As an example of the definition of a class in C++, a very simple description of a class for complex numbers is given.

```
class complex
{
        float re, im;
        complex /*create a complex object*/(float real, float imag);
        complex complex_plus(complex c1, complex c2);
        complex complex_minus(complex c1, complex c2);
        complex complex_mult(complex c1, complex c2);
        complex complex_div(complex c1, complex c2);
        float real_part(complex c1);
        float imag_part(complex c1);
};
```

From this example it should be evident that C++ is syntactically similar to the C language. Note that the class definition specifies both the values (data fields) and the operations (function definitions) associated with the object (a complex number in the example), all of them being called the *members* of the class.† The implementations of the operations can be written in the actual class definition, or (more in accord with the philosophy of separating specification and implementation) can be programmed in a separate file, resulting in a hidden implementation not normally available to the user.

### 2.2.2 Inheritance in C++

C++ supports derived classes, that is, subclasses that share a core of members with its parent class, but adding specific facilities not available in the ancestor or refining already existing ones. Therefore, a class can 'inherit' the features, properties or functionality of a base class without a requirement for re-programming.

† A special member of a class is the *constructor* (whose name is identical to that of the class), which is called automatically anytime an instance of the class is created, thus facilitating a proper initialization of the corresponding data fields.

The following code corresponds to the derivation of a class *circle* from a class *shape*:†

```
class shape
{
    private:
        euclidian_point center;
        font_type font;
    public:
        euclidian_point position() {return center;}
        void translate(euclidian_point new_coordinate) {center=new_coordinate;}
};

class circle: public shape
{
    private:
        float radius;
    public:
        void draw();
        void rotate() {}
};
```

Note that the derived class specifies new data fields and function members not available in the original base class. The members defined in the base class are also accessible from the derived class (in conformity with the visibility rules that are associated with a *public* derivation).

Finally, it should be mentioned that the latest versions of C++ support multiple inheritance.

### 2.2.3 Operator Overloading

C++ provides a mechanism for operator overloading. A programmer can define a special meaning for operators when applied to objects of a specific class.

As an example, consider the complex class. Note that operator overloading provides a more conventional and even convenient notation.

---

† The *private* and *public* specifiers are used to state explicitly the access control that is associated with a given class member. Roughly speaking, a *private* member can be used by member functions of the class in which it is declared. A *public* member can be used by any function. A third option, the *protected* member, can be used both by member functions of the class in which it is declared and by member functions of classes derived from the member's class [ES90].

Of the three currently popular paradigms (logic programming, functional programming, and object-oriented programming), logic programming is arguably the closest to 'natural thinking'†. It deals with very high-level conceptualizations: a program is just a set of specifications. This possibility of writing declarative programs separates program logic from control. It implies two consequences: on one hand, programs are more easily understood and maintained; but, on the other hand, the separation between logic and control can lead to executable programs that are expensive to run. Sometimes, this latter effect has encouraged programming language designers to introduce special features that offer mechanisms for producing more efficient executable code‡.

Since their introduction, logic programming languages have been associated with artificial intelligence problems [Bra86]. Their declarative style and power of expression are desirable characteristics that have made these languages suitable for a wide range of applications, from expert systems applications to database management, and from compiler writing to theorem proving in predicate calculus.

### 2.3.1 Prolog, a Logic Programming Language

Prolog [CM81] is the most-widely used programming language based on logic programming concepts. In fact, Prolog derives its name from 'Programming in Logic'. However, in order to keep it practical, some compromises had to be introduced into the language design and, as a result, Prolog is not considered to be a pure logic programming language.

Prolog is an *applicative* language in the sense that variables can be bound at most once along an execution path, and a *nondeterminate* language since several alternative paths can be explored to obtain a consistent set of variable bindings.

A Prolog program consists of a collection of logic clauses (often referred to as *rules* and *facts*). The user can then formulate some questions (that is, *queries* or *goals*) using those clauses.

### 2.3.2 Prolog Terms

Prolog's data objects are called *terms*. A term can be either simple or compound. A simple term is either a constant number (integers and real are the most commonly implemented), a variable, or an atom standing for itself. A variable is an object that can be bound to another term only once. A special case is the so-called *anonymous* variable which is one that appears only once in a clause.

---

† In the sense of 'common-sense reasoning' (cf. [Prz90]).

‡ Prolog's cut is one example of such "impure" control constructs.

A compound term or *structure* consists of an atom followed by a parenthesized sequence of *n* subterms. The atom is called a *functor* of *arity* n and the subterms are called the *arguments* of the functor. In fact, an atomic constant is equivalent to a structure with zero arity.

There are several proposed notations for Prolog. In this thesis we use the Edinburgh syntax [CM81]. With this version of Prolog, an atom must begin with a lowercase letter (e.g., true, tree), whereas a variable name must start with an upper-case letter or an underscore '_' (e.g., Variable, TREE, _value). Typical examples of structures are: goal(Arg1,Arg2), greater(X,Y,Greater), day(monday). Normally, a structure is referred to using the notation name/arity as in goal/2 or greater/3.

Since lists are heavily utilized in code, a special status is granted to lists in Prolog. The standard functor 'name' for the list constructor is the dot symbol '.'. The Edinburgh implementation also allows lists to be represented using square brackets, for example, [Head|Tail], [1,2,3,4]. The null list is represented by the symbol [ ].

### 2.3.3 Prolog Clauses

Relations in Prolog are specified by means of *clauses*. In Prolog, such clauses are expressed in the form of 'Horn clauses' [Hor51], that is, clauses having the following general form:

$$P \text{ if } Q_1 \text{ and } Q_2 \text{ and } \ldots \text{ and } Q_n$$

or in Prolog notation,

$$p \text{ :- } q1,\ q2\ \ldots\ qn$$

p being the *head* of the clause and the conjunctive part being the *body* of the clause.

The head of the clause is either an atomic symbol or a structure. The atomic symbol or functor is said to be the *predicate* symbol of the clause.

The body of the clause is a sequence of terms separated by commas. Basically, each body term is either an atomic goal or structure.

For example, the predicate

```
dates(gertrude, X) :- male(X), likes(gertrude,X).
```

can be interpreted as: "*Gertrude dates a person if this is a man and Gertrude likes him.*" A special case of these clauses occurs when the conditional part is non-existent, and the rule degenerates into a *fact*, which, furthermore, can be required to be a variable-free

atomic formula. A clause without a head is referred to as a *query*, and is evaluated immediately.

Typical examples of facts are:

```
male(felix).
square(5,25).
```

These facts just record unconditional valid values: "*Felix is a man*", "*The square of 5 is 25*". A collection of facts is often referred to as the *database* that is used to solve a particular problem.

A simple query example is:

```
:-dates(gertrude,X),age(X,Age),Age<30.
```

which can be interpreted as a request for possible persons for Gertrude to date, where each person has an age less than 30 years. (The *age* relation must be able to give the ages of all men in the database).

Prolog semantics can be viewed declaratively or procedurally [SS87]. Under the *declarative view*, a program is considered to be a logical disjunction of its constituent clauses; a clause is viewed as a logical conjunction of its goals.

The *procedural view* considers a program to be an ordered sequence of entry points that must be executed in turn until one of them succeeds; a clause is viewed as an ordered sequence of procedure calls, all of which must be executed in order to achieve success. If failure takes place on any procedure call, the computation must resume at the entry that corresponds to the most recently invoked procedure which contains unattempted entry points waiting to be explored.

### 2.3.4 Searching Strategy in Prolog; Unification and Backtracking

Prolog's searching mechanisms are now discussed in more detail. Given a specific query, Prolog tries to satisfy it. If a valid answer can be discovered, Prolog answers *yes* and shows the values for variables used in the query that satisfy the query. Since Prolog deals with relations, it is always possible for the query to have more than one answer, each one of them being displayed after the explicit request from the user (usually by typing a semicolon after a valid solution has been generated by Prolog). Prolog answers *no* to a query if it fails to satisfy that particular query or no more solutions remain to be displayed.

Logic programming in itself does not specify any evaluation strategy. Prolog, however, assumes a left-to-right and a depth-first search procedure.† Thus, control in Prolog is characterized by two simple decisions:

(1) Goal order: choose the left-most sub-goal.

(2) Rule order: select the first applicable rule.

The process of satisfying a query involves two important operations: *unification/substitution*, and *backtracking*.

*Unification* is a pattern-matching operation that obtains the most general possible common instance for any two Prolog terms. It is used to build and access compound terms and to bind variables to values that satisfy the clauses. The associated *substitution* operation is a mapping from variables to terms, that is, every variable is assigned to a term when unification takes place. If there is no set of substitutions that can make two terms identical the unification fails.

If, while satisfying the clauses and finding the values for the variables that make them true, it is discovered that, at some point, the query cannot be satisfied (that is, unification fails), a process of *backtracking* must be launched. It must return to a point where alternative solutions can be tried (a choice point), restoring all variable bindings to the state they had when the choice point was activated.

### 2.3.5 Extra-logical Features in Prolog

In order to achieve efficiency and facilitate the programming task, Prolog provides some extra-logical features. They include many built-in predicates and the cut operator.

Some of the built-in predicates provide system facilities (such as input/output, file handling), or facilitate the evaluation and comparison of expressions. It is important to note that some of the built-in predicates may have side-effects (which are strange to pure logic programming).

Prolog also provides facilities to control backtracking. The most important of these predicates is the *cut* operator (represented as the exclamation mark symbol in the Edinburgh syntax). The insertion of a cut in the scope of a predicate has two effects when executed. First, all the choices made since the parent goal was unified with the head of the clause in which the cut appears are committed. Second, all other alternatives after this point are completely discarded. This predicate has advantages concerning the

---

† The procedural meaning of Prolog is based on the *resolution principle* for mechanical theorem proving. Prolog uses a special strategy for resolution theorem proving called SLD [Bra86].

reduction of the search space and, therefore, the possibility of finding more efficient Prolog programs.

Another impure (under a pure logic programming point of view) feature of Prolog is the capability to add or remove clauses at run-time through primitives such as *assert* and *retract*. This feature supports dynamic code.

# Chapter 3. An Overview of The Warren Abstract Machine

Since Prolog was introduced in 1972, several abstract architectures have been proposed to support the implementation of the Prolog programming language. There are two common approaches, the environment stacking model and the goal stacking model [Tic88]. However, the environment stacking model has proven to be superior to the goal stacking approach, mainly because it requires fewer run-time tests and usually consumes less (stack) memory while performing resolution.

Amongst the several abstract machines that have been proposed for Prolog, one has become the essential starting point for writing a Prolog compiler: the Warren Abstract Machine (WAM) [War77] [War83]. It is an ingenious design which makes the application of some useful optimizations to the generated code straightforward. Thus, typical Prolog optimizations such as tail-recursion optimization or clause indexing fit naturally into the WAM execution model. This chapter is almost entirely devoted to the Warren Abstract Machine. Particular emphasis is devoted to the description of some optimization concepts within the WAM, namely last call optimization (a general case of tail recursion optimization), environment trimming and clause indexing.

## 3.1 The Warren Abstract Machine (WAM)

D.H.D.Warren proposed the DEC-10 Prolog abstract machine in 1980 [War80] [Ait91]. Its successor, the so-named "Warren Abstract Machine" (WAM, for short) was created in 1983. Both execution models for the Prolog language share the characteristic that their instruction sets correspond closely to the Prolog source code.

In the rest of this chapter, the *procedural view* for Prolog is adopted, since it is easier to relate the WAM's terminology to a model that considers the existence of mechanisms such as procedure call, success, failure, backtracking, and so on (concepts that are absent from the declarative view).

To simplify the study of the WAM, we start with a summary of the memory layout and registers which compose this abstract machine.

### WAM's Memory Layout

The WAM defines at least five distinct working areas: the code area, the heap, the stack, the trail, and the unification stack. The *heap* is a dynamic area intended to store Prolog terms. The *stack* area is designed to contain very specific (data and control) information that is needed to characterize a particular *state* at an arbitrary point in the execution of a Prolog program. The WAM's stack contains two kinds of variable-length stack frames: *environments* and *choice points*. Generally speaking, an environment contains

local variables and book-keeping information, whereas a choice point holds arguments passed to a nondeterminate procedure and backtracking information. These two structures are studied in more detail in future sections.

**Variable Registers**

The WAM assumes the existence of a "sufficient" number of registers to hold Prolog terms during code execution. The principal advantage of using registers instead of heap or stack cells is that objects allocated in registers can be accessed faster and more easily than those stored in memory areas. Arguments passed to a procedure call are always stored in variable registers (often referred to as *argument* registers).

## 3.2 WAM Model Calling Mechanisms

Before starting a thorough analysis of the WAM's instruction set, it is essential to understand WAM's underlying philosophy concerning calling conventions and resolution strategies.

WAM's calling convention is as follows. The caller, that is a body-clause (goal) structure with $n$ terms or arguments, loads all its arguments into dedicated argument registers in textual order and allocates a new *environment* onto the stack; control is then passed to the callee (invoked predicate). The callee selects the different clauses to be tried by means of *indexing* instructions. If the callee is non-determinate (i.e., if the resolution strategy needs to consider more than one clause for that predicate), a *choice point* must be generated and stored on the environmental stack, along with the argument registers and the state registers needed to perform backtracking if necessary. For every plausible head for the callee, *unification* of the arguments is attempted; if such unification is successful, the different goals of the corresponding body are called one after the other in sequential order.

If unification fails at any point, a process of *backtracking* is launched in such a way that the state needed to search for alternative solutions is restored from the current choice point; if no alternatives remain, the choice point is removed and the previous choice point is used to find other potential alternatives.

The WAM classifies variables into two categories: permanent and temporary. By definition [Ait91], "a *temporary* variable is one which does not occur in more than one body goal (counting the head as a part of the first body goal) and first occurs in the head, or in a structure, or in the last goal. A *permanent* variable is one which is not temporary." The motivation for this division is that "temporary" variables need to be explicitly allocated on the heap, whereas "permanent" variables can safely be allocated within an environment frame and be destroyed along with the frame at deallocation-time.

Stack ("permanent") variables are carefully ordered at the end of the frame with one purpose in mind: the space allocated to variables that are no longer needed at some point can be reused. Thus, the environment frame will normally shrink during the execution of a query. Obviously the variables that cease to be needed first during query execution should be stored nearer the end of the frame, so that *environment trimming* can be performed easily.

To behave in accordance with the above-mentioned calling mechanisms the WAM defines a group of control instructions for allocation and deallocation of stack frames, which are named allocate and deallocate, respectively.

The instructions that manipulate choice points perform two fundamental actions: (1) they allocate a new choice point on the stack, and (2) set the order in which the generated code is going to be executed. These generic WAM instructions are named: try_me_else, retry_me_else and trust_me, depending on whether the clause under consideration is the initial clause, an intermediate clause, or the last clause, respectively.

### 3.3 Term Representation

After having studied some of the WAM control and choice instructions that deal with backtracking mechanisms, we now look at those instructions designed for implementing unification. To understand unification, it is essential to know how Prolog's terms are represented.

A term is represented by means of two fields: the tag and the actual value. The *tag* field specifies one of the following possible cases: constant, variable, structure, or list. The WAM design recognizes that lists are heavily used in Prolog and, therefore, they are assigned a special status.

The *value* field contains information specific to the nature of the term. For a constant, it holds the actual value of the constant. In the case of an unbound variable, a structure or a list, the field contains the address of the entity (the arbitrary representation for an unbound variable is a pointer that references itself). A structure value is composed of a representation of the functor (composed of both the name and arity) followed by its arguments (each of which is a term). When a variable is bound to a structure, list or constant, the value field carries the address of the entity to which a variable is unified to. Finally, when a variable is unified with another variable, one of them points to the other (depending on very specific rules that determine the direction of the binding).

A nice property of the WAM architecture is that any term can be stored in a processor register if convenient, independently of the data type of the term.

### Term Construction and Recognition

Now that the representation of Prolog terms has been covered, the process of building and identifying such terms is now analyzed. As expected, the WAM's instruction set defines special instructions to store new terms on the heap and to retrieve such information.

During the process of constructing terms on the heap, it is convenient to store some values temporarily in processor registers so that they can be referenced later more efficiently. The WAM assumes the existence of a sufficient number of registers which can hold all the term representations mentioned above. There are three kinds of objects that are naturally assigned to variable registers: (1) the different arguments of every structure (which under the procedural view of Prolog semantics constitute the parameters of a procedure call); (2) the non-anonymous variables occurring in every rule, query or fact; and (3) the structures and lists that occur inside another structure. For a specific term, the WAM specifies a unique order for assigning the variable registers. The registers are assigned first to the arguments of the procedural call, and only then are the variables and inner structures assigned. All these entities are assigned according to the textual order in which they appear; the same register is assigned to all the occurrences of a particular variable.

The WAM provides two kinds of instructions for manipulating terms: those that require the term to be held in a register and those that work directly on the term under consideration (normally allocated on the heap or on the stack).

As has been mentioned before, the arguments of a call and some other special terms (variables and the "inner" structures) are always assigned to registers. The WAM provides two sets of instructions to manipulate those entities: the *put* instructions, intended to store in the argument registers the parameters that are passed to a procedure call; and the *get* instructions, designed to access those parameters once the call has been activated.

### Unification

The WAM defines additional term manipulation instructions. Their behaviour depends on the mode set by the *get_structure* instruction. If *read* mode is set, the unification algorithm is applied to both the instruction operand and the current heap cell. If, instead, *write* mode is specified, a new cell is allocated onto the heap.

Two examples that illustrate typical translations of facts (head clause) and queries (body clause) are shown in Figures 3.1 and 3.2. The corresponding register assignment is shown in Table 3.1.

predicate/3 :

```
get_variable X0              % predicate(
get_structure m/2,X1         %   V,
unify_variable X5            %   m(
unify_variable X6            %      X5,
get_structure n/2,X2         %      W),
unify_value X0               %   n(
unify_value X6               %      V,
get_list X5                  %      W))
unify_constant a             % X5=[
unify_variable X4            %      a|
get_list X4                  %      X4]
unify_constant b             % X4=[
unify_variable X3            %      b|
get_list X3                  %      X3]
unify_constant c             % X3=[
unify_constant [ ]           %      c|
                             %      [ ] ]
```

```
predicate(V,m([a,b,c],W),n(V,W))
```

**Figure 3.1** Partial translation of a clause head.

| Register | Type | Term |
|---|---|---|
| X0 | Arg 1 | V |
| X1 | Arg 2 | m/2 |
| X2 | Arg 3 | n/2 |
| X3 | LIST | [ c \| [ ] ] |
| X4 | LIST | [ b \| X3 ] |
| X5 | LIST | [ a \| X4 ] |
| X6 | VARIABLE | W |

**Table 3.1** Register assignment for the examples in Figures 3.1 and 3.2.

```
put_list X3                 % X3=[
unify_constant c            %  c|
unify_constant [ ]          %  [ ]]
put_list X4                 % X4=[
unify_constant b            %  b|
unify_variable X3           %  X3]
put_list X5                 % X5=[
unify_constant a            %  a|
unify_variable X4           %  X4]
                            % predicate(
put_variable X0             %  V,
put_structure m/2,X1        %  m(
unify_variable X5           %    X5,
unify_variable X6           %    W),
put_structure n/2,X2        %  n(
unify_value X0              %    V,
unify_value X6              %    W))
```

```
:- ..., predicate(V,m([a,b,c],W),n(V,W)), ...
```

**Figure 3.2** Partial translation of a query.

### Last Call Optimization and Environment Trimming

Two very important optimization techniques that are applicable to Prolog's code base their operation on a more rational use of the environmental stack, namely last call optimization (LCO) and environment trimming.

*Tail recursion optimization* (TRO) is a widely-used optimizing transformation that converts recursion into iteration [AC72]. Under certain conditions, if the last action that a procedure performs is a recursive call (that is, a call to itself), this *tail-recursive* call can be uniformly encoded using *jump* instructions, thus speeding up the process and saving the allocation of a new stack frame. A particular effect of this optimization is that tail *recursion* (which normally consumes a stack frame for each call) is transformed into *iteration* (which ideally operates in constant space).

*Last call optimization* permits re-use of part of the current environment frame (but only if it is at the top of the stack) when the last goal of a clause is invoked. This technique is based on the fact that just before the last call of a query, none of the stack variables is needed any longer, and therefore their space can be overwritten by that last call. *Tail recursion optimization* is a special case of last call optimization, namely when the last call happens to be a *recursive* call.

Furthermore, memory can be recovered on the environmental stack before *any* call, provided that one or more stack variables are used subsequently. This may be viewed as a generalization of last call optimization and is called *environment trimming.*

In order to apply environment trimming, it is necessary to rearrange the order of the variables within the environment frame: those variables which cease to be required first are allocated nearer to the end of the frame and those that are used closest to the end of the query are assigned to the initial slots of the environment. As an example, consider the predicate of Figure 3.3. Note that whenever a variable is no longer needed, its memory location is freed for future use. Last call optimization takes place when, at the very last call, none of the variable locations of the environment is preserved. These locations may be released for future use.

### 3.4 Clause Indexing

This last section is devoted to the study of a non-trivial optimization technique called clause indexing. The incorporation of this fundamental notion into the WAM represents the last stage in this brief tour through the WAM.

In Prolog, when a body subgoal is attempted, if the called predicate is nondeterministic, several clauses are candidates to be tried in the search for a successful path. By using *clause indexing* techniques, the number of attempted clauses can be reduced. The key idea is that, given a predicate query whose arguments have particular bindings at the time of the call, some head unifications are guaranteed to fail because of an incompatibility between those bindings and the kinds of terms that are specified in the heads of the clauses. An example of clause indexing is shown in Figure 3.4.

Given the instantiation of a call argument ("body" argument) and the term category of the corresponding argument in the head of the predicate clause ("head" argument), three possibilities can occur: (1) the arguments unify without any restriction (because at least one of them is an unbound variable), (2) unification fails (as a result of having incompatible categories), or (3) the unification takes place if some additional conditions are met (for example, if the two arguments are constants, their values must be identical for unification to occur). This is summarized on Table 3.2. It is obvious from the table that a variable body argument can match every head argument; other term categories only match head arguments that are either unbound variables or are derived from the same term category as the body argument.

Thus, it is possible to determine argument-wise which clauses are matchable for a given calling pattern. For this to occur, all the body arguments must be totally compatible with the corresponding head arguments. However, it is a standard practice to test only one argument, the first argument. This practice is less expensive than testing all

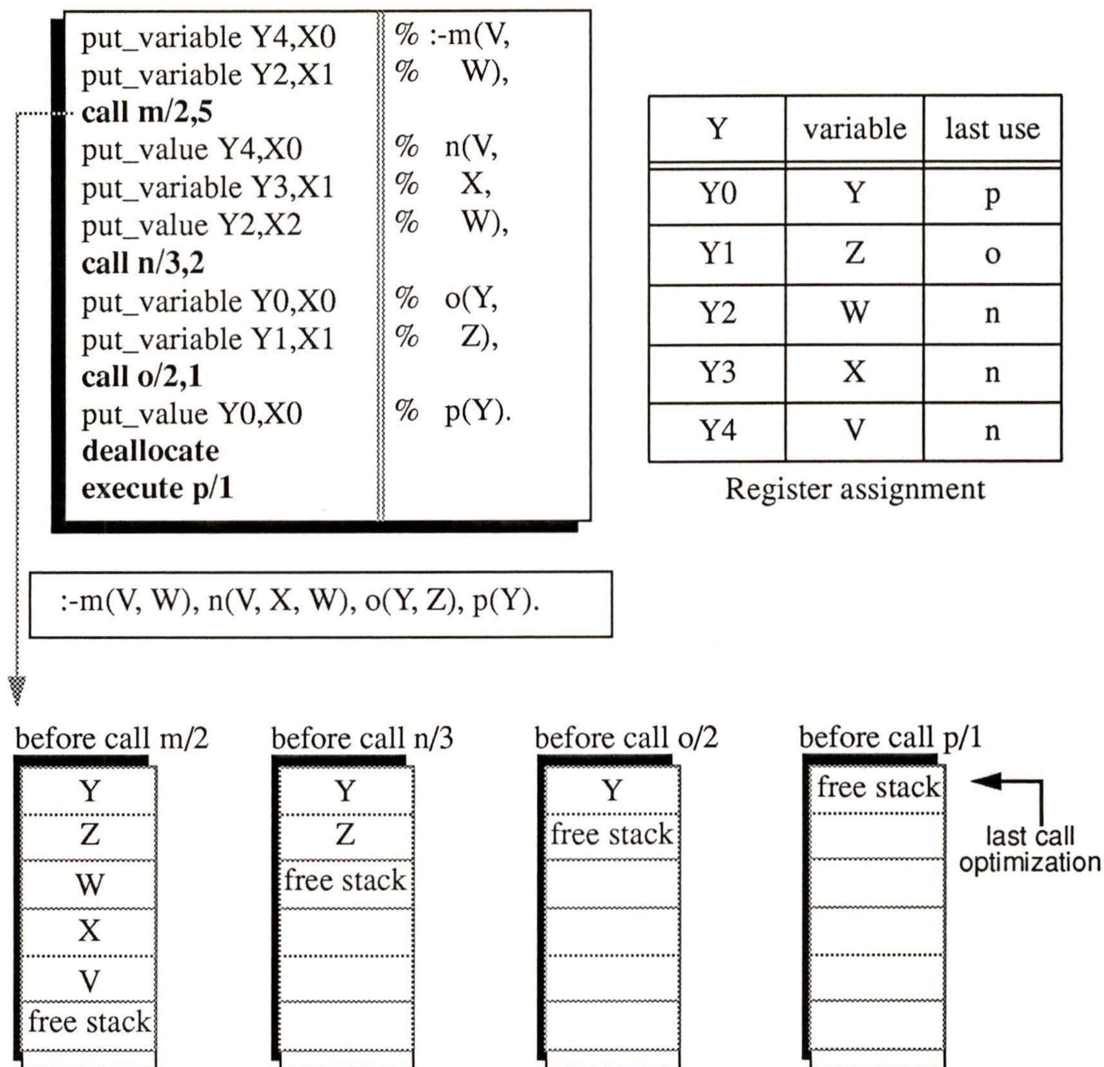



**Figure 3.3** Environment trimming.

the arguments, and it can be argued that a good number of predicates can be easily discriminated by the nature of their first argument [Ait91].

This single argument indexing is adopted by the WAM. Since a *variable* first argument in a head clause unifies with any body's first argument, the translation is split over subgroups of clauses. Each subgroup is either a single clause with a variable first argument or the maximal possible group of contiguous clauses whose first argument is not a variable.

The general translation for a predicate partitioned into subgroups of clauses $S_1,...,S_n$ is shown in Figure 3.5.

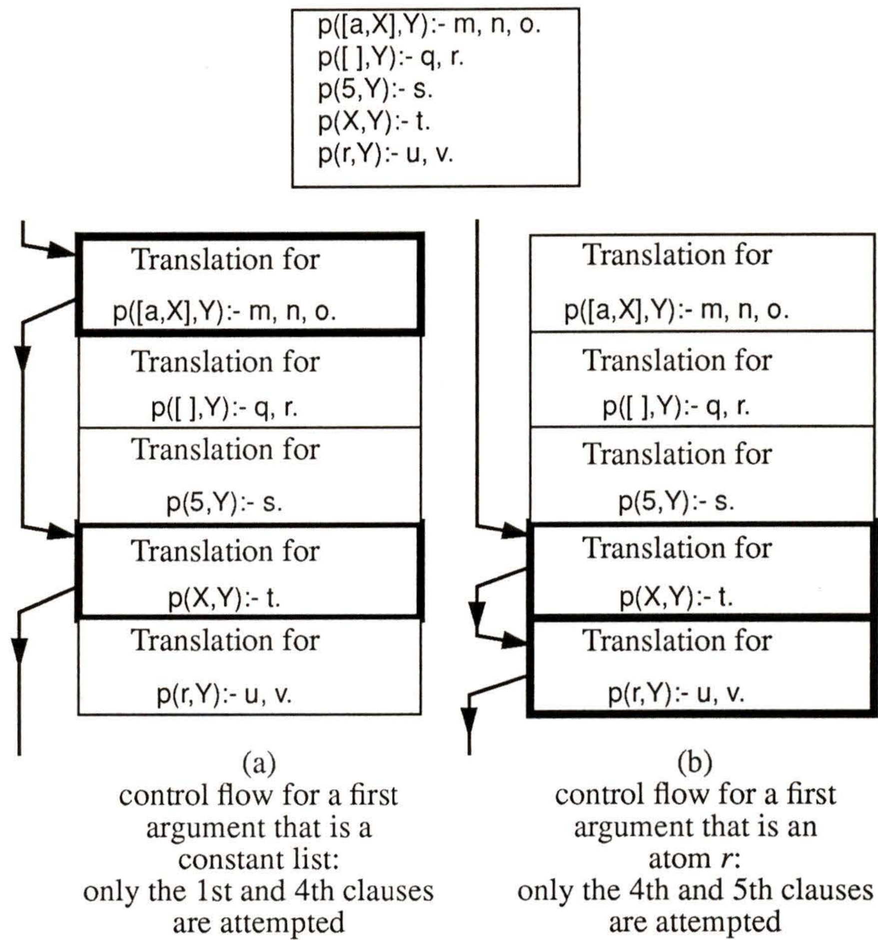



**Figure 3.4** An example of clause indexing.

| | | |
|---|---|---|
| predicate: | try_me_else S2<br>S1's indexed code | % subgroup of clauses S1 |
| S2: | retry_me_else S3<br>S2's indexed code | % subgroup of clauses S2 |
| | ⋮ | ⋮ |
| Sn: | trust_me<br>S1's indexed code | subgroup of clauses Sn |

**Figure 3.5** The general translation scheme for predicates.

When the subgroup is constituted by a single clause with a first argument variable, standard translation patterns are applied. However, if the subgroup is formed by one or

head argument

| body argument | variable | constant | structure | list |
|---|---|---|---|---|
| variable | UNIFY | UNIFY | UNIFY | UNIFY |
| constant | UNIFY | POSSIBLE | FAIL | FAIL |
| structure | UNIFY | FAIL | POSSIBLE | FAIL |
| list | UNIFY | FAIL | FAIL | POSSIBLE |

two arguments are FULLY UNIFIABLE if the following holds:
- for constants: both constants have the same constant value
- for structures: both structures have the same functor, the same arity and fully unifiable arguments
- for lists: both lists have compatible arguments

**Table 3.2** Different possibilities for the unification of two terms.

more clauses with non-variable first arguments, specially tailored instructions must be used.

The technique of clause indexing that is used in the WAM machine can be better understood when it is decomposed into the so-called *levels of indexing*. Some authors [Ait91] acknowledge the existence of three levels of indexing. The first two levels have to do with the selection of the *first* unifiable clause for a given (first) argument, and the third level corresponds to the selection of the *next* unifiable clauses for that argument.

*First level* indexing simply discriminates clauses according to the type of the first argument in the query. Thus, depending on the nature of the first argument (variable, constant, list or structure) only those clauses that are unifiable with such an argument are considered. *Second level* indexing further discriminates clauses according to the specific values that are associated with constants and structures. Finally, *third level* indexing is aimed to find out all next clauses that must be attempted once the first clause has provided all its answers (if any).

# Chapter 4. The TOPIC System

The TOPIC system is a software application that supports the multi-programming paradigm by allowing programmers to combine Prolog and C++ code in the same program. The TOPIC system provides both a *translator* from Prolog into C++, and a *library* containing those C++ classes that are used in the translation scheme of the TOPIC system. The TOPIC system is extensively described in a series of documents [JLH90] [Jun90a] [Jun90b] [Jun90c] [Jun90d] and, therefore, only the most important elements are discussed here.

The TOPIC system is described in two parts. An initial section explains the constituent elements of the TOPIC system. The concluding sections explore how the TOPIC system accomplishes the translation of Prolog into C++.

## 4.1 Introduction to the Topic System

The TOPIC system provides a Prolog to C++ translator, along with a library of primitive C++ classes that are needed for the implementation of the Prolog language in terms of C++. The Prolog-to-C++ translator converts Prolog source code into equivalent code that is built in terms of C++ classes. Conceptually speaking, the translator can be separated into a syntax analyzer, a semantic analyzer, and a code generator.

### Syntax Analyzer

The TOPIC syntax analyzer is tailored to the Edinburgh Prolog syntax [CM81]. It contains a lexical analyzer, a special operator filter, and a standard parser. Following traditional compiler techniques, the user-defined identifiers, such as atoms, functors and operators, are stored in an *identifier table*. In order to achieve correct handling of Prolog operators, whose precedence can be specified by the user, a group of *operator table*s is built to hold the dynamic properties of the operators (i.e., position, precedence, associativity).

When an operator token is found, the analyzer looks up its characteristics in the operator tables. The module that conveys this action is the *operator filter*, whose sole function is to identify the characteristics of an operator token given a context (which is recorded in the operator tables).

Finally, a standard *parser* is needed to verify that the source code satisfies the rules of the Edinburgh Prolog syntax and generates the necessary structures for subsequent phases of the translator. Again, syntax errors are reported to the user via an error manager module.

**Semantic Analyzer and Code Generator**

The analysis phase is completed by a semantic analyzer which tests that the semantic rules of Prolog are not violated by the syntactic constructs previously analyzed by the parser. Besides this process of verification, the semantic analyzer must generate information for the code generation module.

The semantic analysis is performed at the Prolog clause level, i.e., the basic unit that is taken into account during analysis is a Prolog clause. Four different kinds of clauses are accepted: TOPIC directives, goal clauses (queries), complete rules, and facts.

The code generator must convert each Prolog predicate into a C++ subclass definition. The translation scheme is discussed in the following section. At this point, we need say only that the generated code references a set of C++ classes that are defined and implemented in the TOPIC library. A fairly complete description of the code generation phase can be found in the document [Jun90b].

## 4.2 The TOPIC Translation Scheme

In this section, the translation strategies used by the TOPIC system to convert Prolog source code into C++ code are examined. Initially, the strategy used by the TOPIC system to perform the Prolog control flow is discussed. Thereafter, an overview of the memory organization in the TOPIC system is presented. Finally, the most important issues relevant to the translation of Prolog into C++ with the TOPIC system are analyzed.

### 4.2.1 Control Flow

To understand how the TOPIC system implements Prolog's control flow, it is useful to represent the operational semantics of Prolog schematically. By using *box* diagrams [Byr80], the depth-first left-to-right evaluation order used in the execution of Prolog programs can be easily described.

**Subgoal Representation**

Consider a Prolog clause

$$h \text{ :- } c_1, c_2, ..., c_n.$$

Every subgoal $c_x$ can be represented by a box like that shown in Figure 4.1. The box has two entries (labelled 1 and 3) and two exits (labelled 2 and 4). The pair of ports 1 and 2 are used when the normal sequential execution takes place; the other ports, 3 and 4, are used when backtracking occurs. Ovals *A* and *B* represent the search for answers to the subgoal query. When the execution reaches the subgoal box (either through

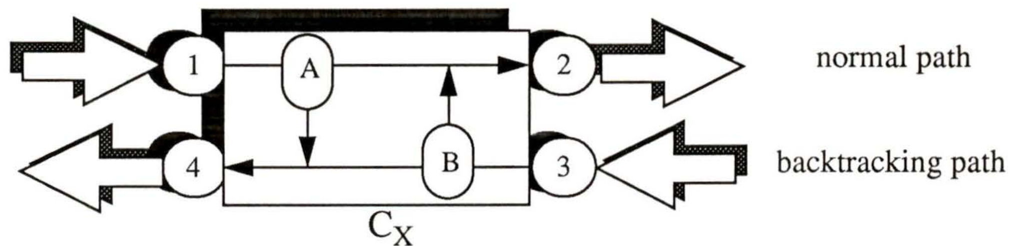



**Figure 4.1** The box representation of a subgoal.

entry 1 or entry 3), if the subgoal succeeds, the execution flow continues through exit 2. Otherwise, when the subgoal fails (that is, when no alternatives remain to be explored for that particular subgoal), backtracking takes place and the execution flow continues through exit 4.

### Query Representation

Consider a query of the form

$$:- c_1, c_2, ..., c_n.$$

A schematic representation of this generic query is shown in Figure 4.2.

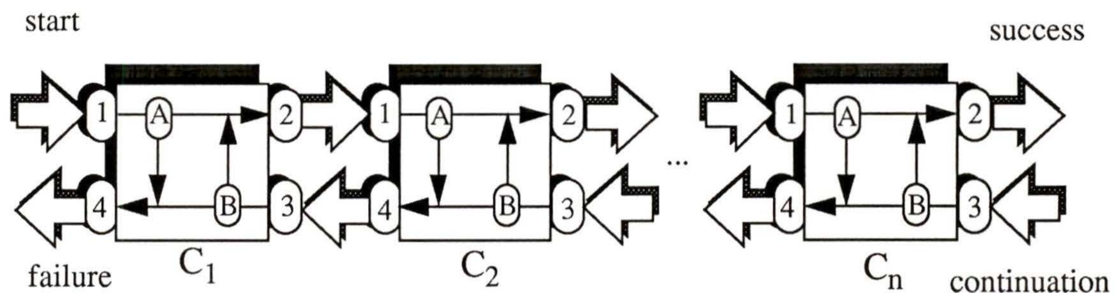



**Figure 4.2** The box representation of a goal.

Individual boxes for each subgoal are superimposed in the obvious way. Entry point 1 of the first subgoal is designated as the *start* point for the query execution. Exit point 2 of the last subgoal represents the discovery of a valid solution to the query; it is denominated as *success*. If more answers are requested, execution must resume through entry point 3 of the last subgoal, labelled *continuation*. Exit point 4 of the first subgoal is special in the sense that if the flow reaches this point, the query is unable to provide more answers; the query *fails*.

### Predicate Control Flow

The box representation for a generic predicate

$$p \text{ :- } c_{11}, c_{12}, \ldots, c_{1j}.$$

$$p \text{ :- } c_{21}, c_{22}, \ldots, c_{2k}.$$

$$\ldots$$

$$p \text{ :- } c_{m1}, c_{m2}, \ldots, c_{mn}.$$

is shown in Figure 4.3. The main addition with respect to a simple query box representation is that the heads of the clauses must be taken into account. Such head unifications are represented by the leftmost circles. When the first clause is attempted, if it is possible to unify its head with the arguments given by the caller, the subgoals of the predicate are considered in the same way as for a simple query. If head unification is not successful, control is passed to the next clause, where, again, head unification is attempted, and so on. Failure is reached only after all clauses have been attempted.

### TOPIC's Scheme for Implementing Prolog's Control Flow

Every Prolog predicate is translated into a C++ class that is derived from the *Predicate* base class. This class provides a constructor, as well as both a data field and a *Boolean Next* method aimed at implementing Prolog's control flow.

Thus, the *Predicate* class defines a specific data field to store the *continuation* point where execution has to resume when more answers are requested for the given predicate. This data field is named *where* and it is assigned integer values that represent the clause numbers (starting with 0) for a given predicate. Furthermore, each predicate contains a method (the *Next* method) which finds the subgoal answers using this flow control point of view. A typical translation scheme is shown in Figure 4.4. The associated box representation for the predicate is depicted in Figure 4.5. In order to show the relationship between the TOPIC's translation scheme and the corresponding box representation, the entry and exit points in the box diagram are reproduced in the C++ pseudo-code as labels to which the control is passed depending on whether or not an answer is found for any subgoal.

## 4.2.2 Memory Organization

In order to execute a goal (query), the TOPIC system requires direct control over the allocation/deallocation of the objects that are created during the goal execution. Whenever it is possible, the TOPIC system treats such objects as *local* objects to the goal. The scope of a local object is that of the goal in which the object is created, and its lifetime is that of the goal which it belongs to.

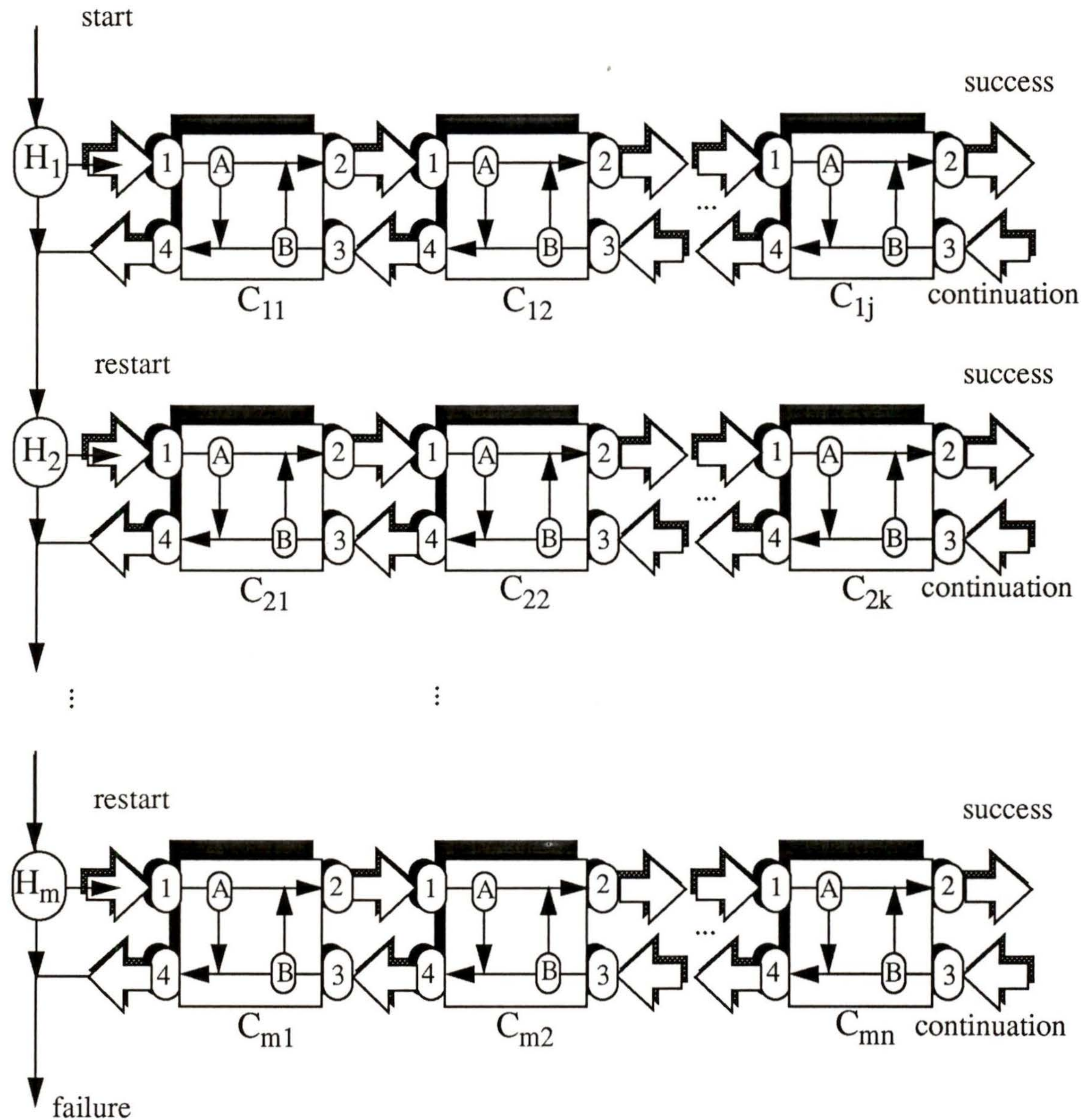
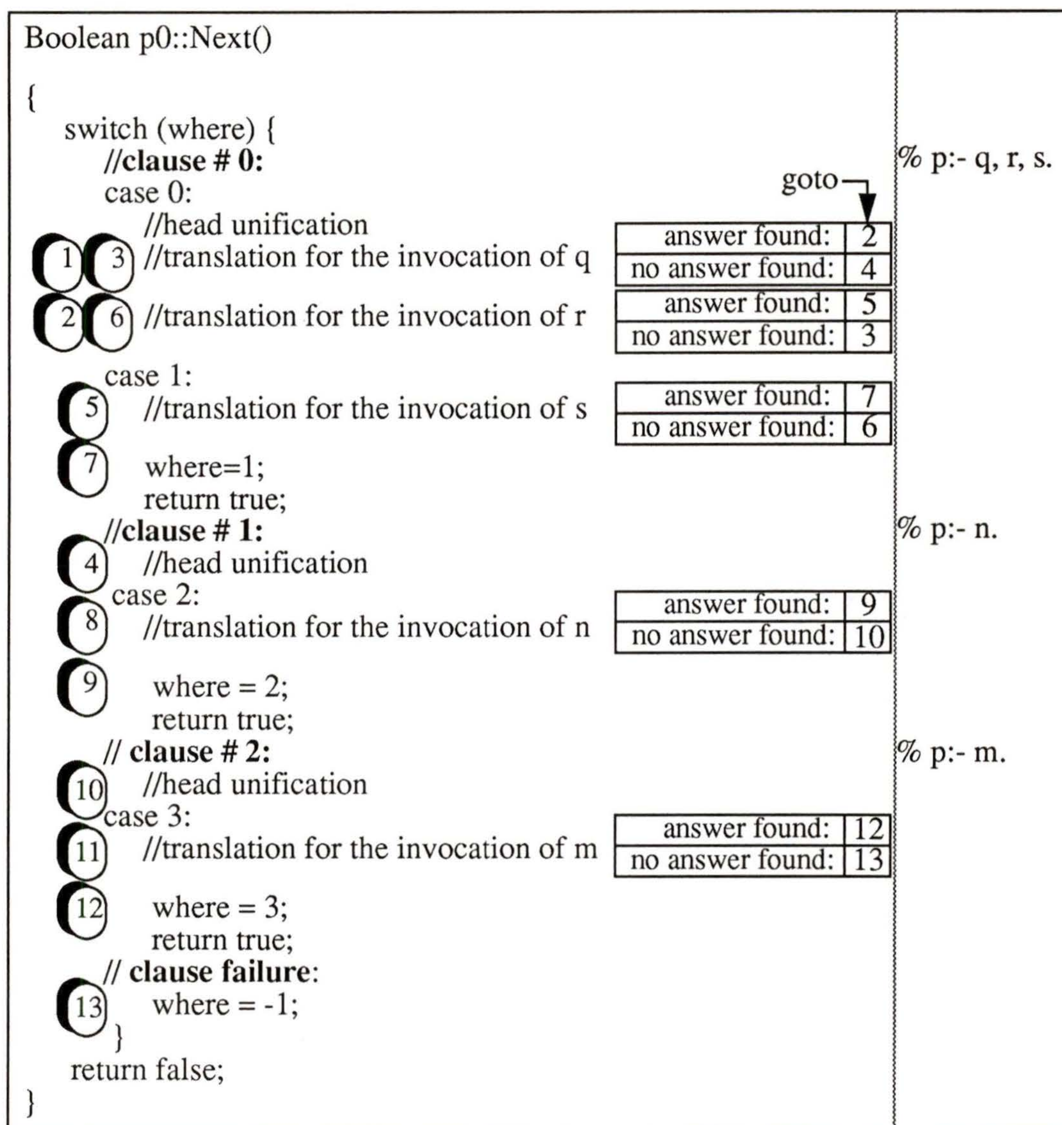



**Figure 4.3** The box representation of a predicate.

However, sometimes it is necessary to extend the scope of an object outside the goal where it is defined. For this reason, the TOPIC system provides mechanisms to create *global* objects (i.e., objects that are not attached to a particular goal). Representative examples of this can be found in the rules that construct a Prolog database.

Local objects are allocated in temporary stacks that only exist while the goal is active. Global objects are allocated in the C++ heap.




The boxes on the right represent branches where the execution will continue depending on whether the invoked predicate was able to provide an answer.

**Figure 4.4** Translation scheme to reflect the flow control of a predicate.

### Working Areas

When a goal is created, the memory assigned to it is partitioned into five stack areas that hold the different entities created during the execution of the goal.

- The Stack. This area holds *Predicate* instances, *State* instances and frames.
- The Heap. This code area contains all *Term* instances (with the sole exception of the *LocalVariable* instances, see below).

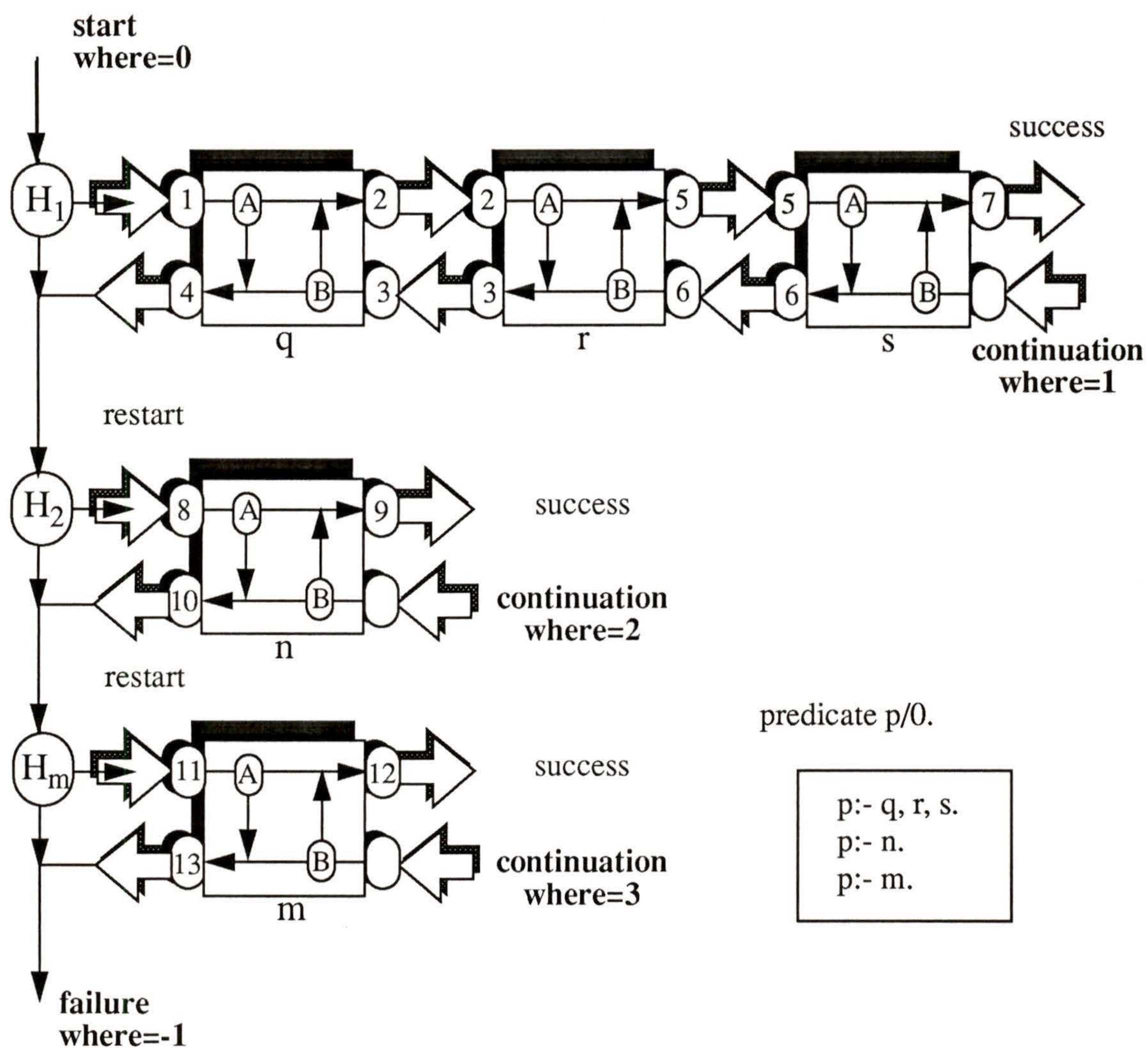



**Figure 4.5** Box representation for a predicate.

- The Trail. The (Variable) Trail contains the addresses of all *LocalVariable* instances whose bindings must be trailed upon backtracking.
- The PTrail. The Predicate Trail holds the addresses of predicate instances so that they can be removed once they are no longer needed by the goal (upon backtracking).
- The GlobalVarHeap. This area holds the goal environment's hash table.

This memory organization is inspired by the WAM. In fact, given the strong similarity between the WAM and the TOPIC system, the memory model for the TOPIC system can be studied from the WAM principles previously introduced.

### Prolog Registers

The TOPIC system does not use registers to store Prolog terms at execution time (terms are always allocated on the heap or stack). However, the TOPIC system provides some state registers (in fact they are declared as static members of the *Goal* class) to manipulate the above-mentioned working areas. As well as the ordinary pointers to the top of the working areas, the TOPIC system provides a state register that points to the latest allocated choice point frame. Additionally, a couple of registers is required to hold the addresses of the current predicate instance of the goal that is being executed and the current goal itself. A simple diagram showing the TOPIC working areas is shown in Figure 4.6.

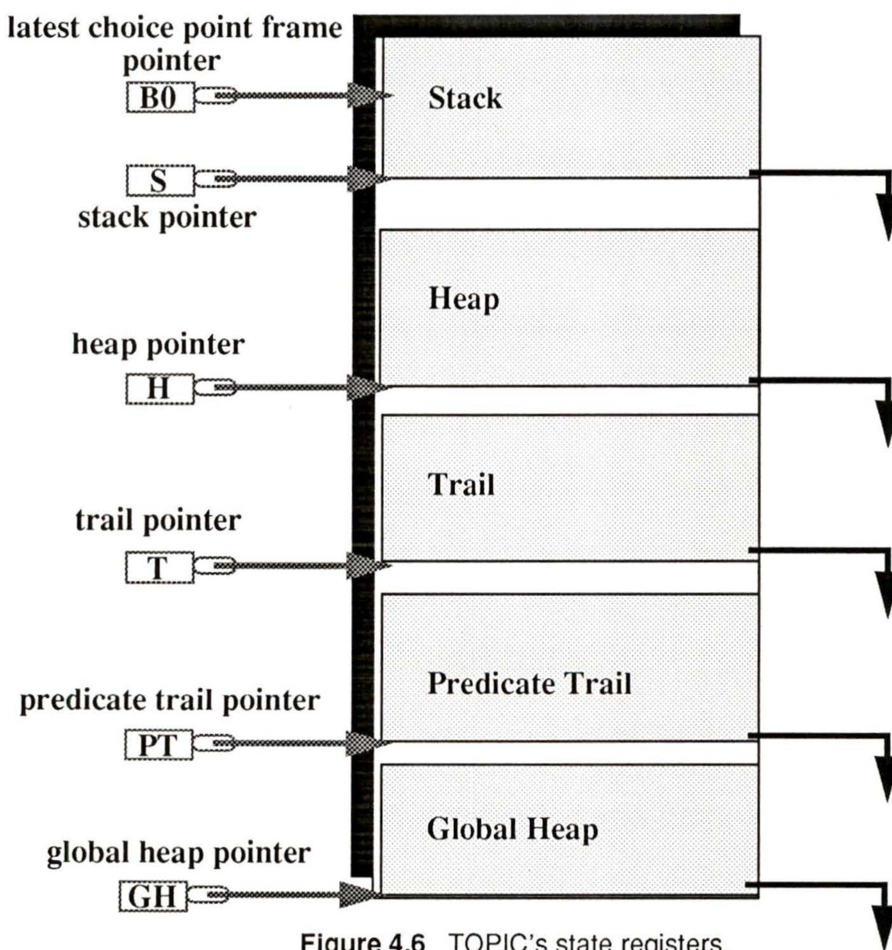



**Figure 4.6** TOPIC's state registers.

## 4.3 Prolog Objects Representation

This section describes how Prolog objects are represented in the TOPIC system using the C++ class facility. The Prolog objects that are studied are the following: terms, queries, heads, rules, predicates and databases.

### Term Representation

Since C++ encourages the use of a hierarchical class definition via step-wise refinements, the TOPIC system adopts this kind of strategy to characterize all different Prolog data objects. Thus, Prolog terms are represented in C++ by using a base class *Term* which is specialized through inheritance into the different categories of simple and compound terms. A hierarchical diagram is shown in Figure 4.7.

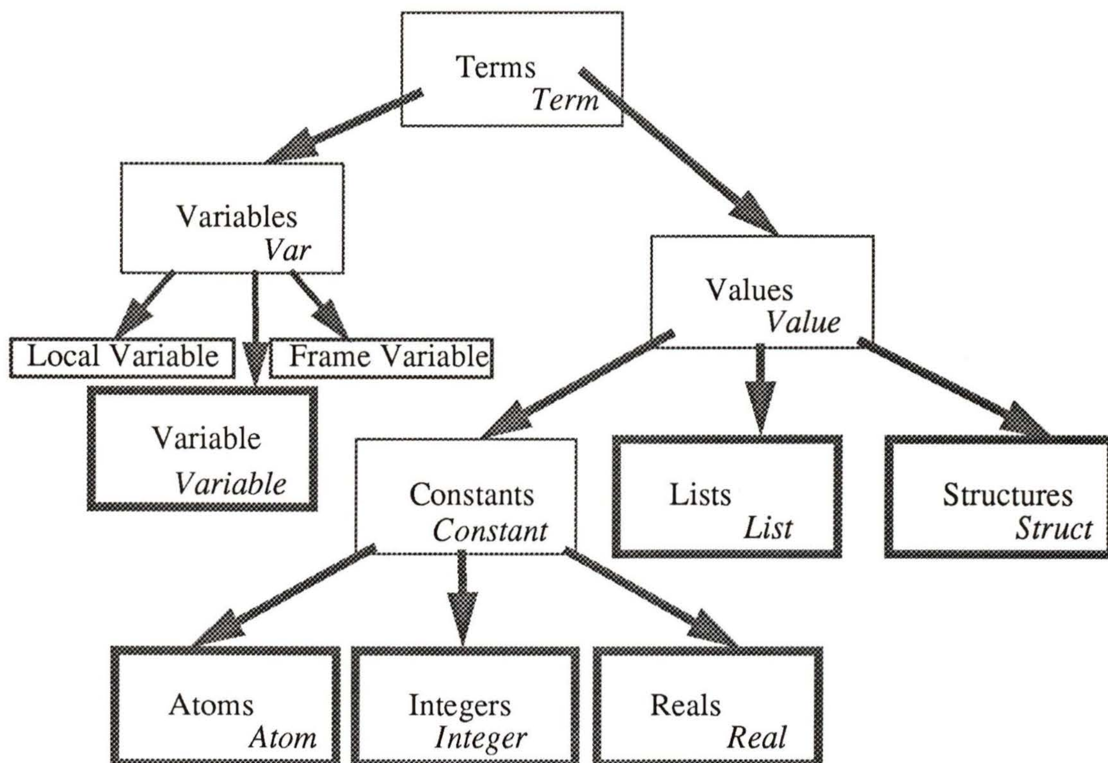



**Figure 4.7** The TOPIC class hierarchy for Prolog terms.

At a first level of specialization, the derived abstract classes representing *variables* (*Var* class) and *values* (*Value* class) are defined. The *Var* class is further specialized into subclasses representing the type of location that is assigned to them, the possibilities being variables declared inside a predicate (*LocalVariable* class), variables allocated in stack frames (*FrameVariable* class), and all other variables (*Variable* class).

The *Value* class is refined to generate the *Constant* abstract class and the *Structure* class. The *Constant* class represents all constant terms and it is additionally specialized into atomic values (*Atom* class) and numbers (*Integer* and *Real* classes). The *List* class is a natural subclass of the *Structure* class, but this dependency relationship is not explicit in the TOPIC system. Instead, the *List* class is defined in terms of the *Value* abstract class. Similarly, it can be noticed that, whereas constants are normally seen as a special case of a structure when the arity is equal to zero, this fact is not reflected in the TOPIC class hierarchy for terms.

In order to create instances of Prolog terms, C++ constructors must be invoked. Note that the TOPIC system is designed to manipulate Prolog terms by means of pointers to their corresponding instances. Some examples of how Prolog terms are created in the TOPIC system are shown in Table 4.1.

| Prolog term | term declaration in the TOPIC system |
|---|---|
| 345 | Term *term1 = new Integer(345) |
| xyz | Term *term2 = new Atom("xyz") |
| f(a,345,xyz) | Term *term3 = new Struct("f",3,new Atom("a"),term1,term2) |
| M | Term *term4 = new Variable("M"); |
| [M,_] | Term *term5 = new List(term4,new Variable) |

**Table 4.1** Term creation in the TOPIC system.

### Clause Representation

The TOPIC system defines C++ classes that represent queries, facts and complete rules. A *Query* class is provided to represent bodies of Prolog clauses. This class definition exploits the fact that every Prolog query can be represented by either a single term, or, in a more general case, by a conjunction, disjunction, implication or negation of terms. For this reason the *Query* class defines its fundamental data field as a pointer to a Prolog structure, and provides a set of friend functions designed to build more complex queries, applying *and*, *or*, *if* and *not* operators to achieve the conjunction, disjunction, implication and negation of queries (respectively). Examples of the usage of this class are shown in Table 4.2.

| Prolog query | query declaration in the TOPIC system |
|---|---|
| :- p. | Query q1(new Struct("p",0)) |
| :- q(X,t). | Query q2(new Struct("q",2,new Variable("X"),new Atom("t"))) |
| :- p, q(X,t). | Query q3(q1 & q2) |
| :- p; q(X,t). | Query q4(q1 \| q2) |
| :- \+p | Query q5(~q1) |

**Table 4.2** Declaration of queries in the TOPIC system.

Similarly, a *Head* class represents heads of Prolog clauses. To represent the head of a clause fully, the functor (name and arity) and the term arguments must be provided. Finally, a *Rule* class permits the construction of complete clauses by concatenating both the head and the query. Some examples of this are presented in Table 4.3.

| Prolog facts and rules | corresponding declaration in the TOPIC system |
|---|---|
| r. | Head h1("r") |
| s(X,t). | Head h2("s",2,new Variable("X"),new Atom("t")) |
| :- q(X,t). | Query q2(new Struct("q",2,new Variable("X"),new Atom("t"))) |
| r:- p, q(X,t). | Rule r1(h1,Query(Query(new Struct("p",0)) & q2) |
| s(X,t):- \+p. | Rule r2(h2,Query (~Query(new Struct("p",0))) |

**Table 4.3** Definition of heads and rules in the TOPIC system.

### Predicate Representation

In the TOPIC system, a predicate is represented using the *Predicate* class. It provides both a constructor and a *Next* method, which is used to get the next available answer for a given query.

Recall that query execution can be viewed as a sequential invocation of the predicates or subgoals that comprise the query. The first time a predicate is invoked, an in-

stance of it is created. The Boolean *Next* method produces the first answer (if any). If more answers are required, the *Next* method remembers the starting point for this future searching as long as more answers remain to be explored. When no more answers can be found, the *Next* method simply returns *false*. The Next method reproduces the control flow behaviour of Prolog that was previously explained using box diagrams. A protected field, named *where*, remembers the specific point where additional attempts to find the next solution should resume.

### 4.4 Unification

Unification of two C++ representations of Prolog terms is performed by means of the Unify member function (defined within the *Goal* class). The usual effect of this function is, if the unification succeeds, to create new variable bindings. It operates within the current goal's context. Its first phase consists of de-referencing both arguments. After this, the new bindings of variables to terms that result from unification are stored within the goal's context.

A special case occurs when the argument of the head to be unified is not a variable. To avoid unnecessary term construction, an alternative member function, Connect, is used. There is one Connect function for each possible kind of Prolog term (i.e., integer, real, atom, list and structure). Figure 4.8 shows how these two functions are used to perform the unification operation.

### 4.5 Backtracking and Frame Allocation

Since the TOPIC system's memory organization follows the WAM model, it is not surprising that the TOPIC system defines several member functions to perform similar actions to those of the instructions for implementing Prolog calling mechanisms in the WAM.

The TOPIC system provides three static member functions specially designed to manage backtracking. They are closely related to the WAM instructions for allocation and deallocation of choice point frames. These member functions (which are declared in the *Goal* class) are: Checkpoint, Backtrack and Restore.

Checkpoint: this function is analogous to the try_me_else instruction of the WAM. It registers all the state information that needs be restored when backtracking is required.

Backtrack: analogous to the retry_me_else instruction in the WAM, this function restores the state that was previously recorded by the Checkpoint function. This function is used for all intermediate clauses in a predicate.

```
                                                         % head unification
                                                         % q(X,Y)
Goal::Unify(_p2c_a1, &X)                                     q(
Goal::Unify(_p2c_a2, &Y)                                         X,
                                                                 Y)

                                                         % r(67,t)
                                                             r(
Goal::Connect(_p2c_a1, 67)                                       67,
Goal::Connect(_p2c_a2, "t")                                      t)

                                                         % p(X,foo(67,t),[ [H|T] ],67)
                                                             p(
Goal::Unify(_p2c_a1, &X)                                         X,
Goal::Connect(_p2c_a2, "foo", 2, &_p2c_t0, &_p2c_t1)             foo(67,t),
Goal::Connect(_p2c_t1.Value(), "t"))
Goal::Connect(_p2c_t0.Value(), 67)
Goal::Connect(_p2c_a3, &_p2c_t2, &_p2c_t3)                       [ [H|T] ],
Goal::Connect(_p2c_t2.Value(), &H, &T)
Goal::Connect(_p2c_t3.Value(), "[ ]")
Goal::Connect(_p2c_a4, 67)                                       67)
```

Note that objects that are labelled "_p2c_a*n*" represent arguments to the predicate and objects labelled "_p2c_t*n*" are intermediate objects that are used to build more complex terms.

**Figure 4.8** Use of the Unify and Connect functions.

Restore: similar to the trust_me operation in the WAM, this TOPIC function also restores the state previously stored by a Checkpoint function invocation. The only difference is that such a state is discarded once it is no longer needed. For this reason, this function is only associated with the last clause of a particular predicate.

A typical example of the use of these functions is shown in Figure 4.9.

Similarly, the TOPIC system defines two member functions that are equivalent to the WAM instructions for allocation and deallocation of environment frames. These member functions (declared in the *Goal* class) are: Allocate and Deallocate.

Allocate: this function is used to allocate a *frame* to contain a predicate's "permanent" variables (recall that a permanent variable is that which can be allocated within an environment frame, being destroyed when the predicate is deallocated).

Deallocate: this function is used to remove a frame from the environment.

## 4.6 Cut Operation

The inclusion of the cut operation in the box model for control flow of Prolog programs is as simple as modifying the structure of the box that is associated with the cut predi-

| | |
|---|---|
| Boolean p0::Next()<br>{<br>switch (where) {<br>//clause 0<br>case 0:<br>**state=Goal::Checkpoint();**<br>//q predicate creation<br>//invocation of q->Next()<br>//r predicate creation<br>//invocation of q->Next()<br>//s predicate creation<br>case 1:<br>//invocation of q->Next()<br>where=1;<br>return true;<br>//clause 1:<br>**Goal::Backtrack(state);**<br>//n predicate creation<br>case 2:<br>//invocation of n->Next()<br>where = 2;<br>return true;<br>// clause2:<br>**Goal::Restore(state);**<br>//m predicate creation<br>case 3:<br>//invocation of m->Next()<br>where = 3;<br>Remove();<br>return true;<br>// clause failure.<br>where = -1;<br>}<br>return false;<br>} | <br><br><br>%p:- q, r, s.<br><br><br><br><br><br><br><br><br><br><br><br>%p:- n.<br><br><br><br><br><br><br>%p:- m. |

**Figure 4.9** Backtracking management for a typical Prolog predicate.

cate. First of all, if backtracking reaches the cut predicate, it will not give more answers. Furthermore, all the choices made since the parent goal was unified with the head of the predicate clause must be committed, and alternatives not yet explored must be discarded. The box model for the cut predicate is presented in Figure 4.10. Note that the cut predicate always succeeds the first time and, when it is reached during backtracking, it will transfer control directly to the point labelled *failure*.

The TOPIC system performs the cut operation with the help of special functions that not only reflect this particular behaviour, but also release the storage used by alternatives that are no longer needed. These functions are NeckCut, BodyCut and TailCut,

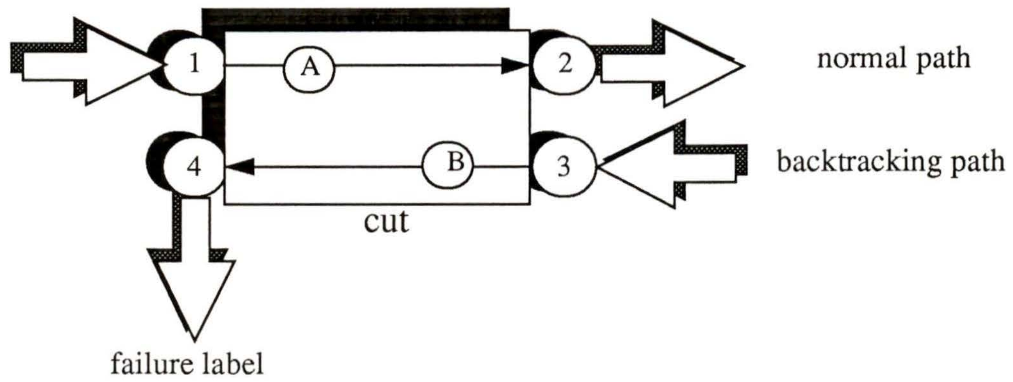



**Figure 4.10** The box representation of the cut predicate.

which are used, respectively, when the cut appears as the first subgoal, as an intermediate subgoal or as the last subgoal of a clause.

# Chapter 5. Mode analysis

In general, Prolog programs are undirected, that is, there is no distinction between input and output parameters for a given predicate. This notion of bi-directionality presents a major challenge to the production of efficient code, since the depth-first search strategy with chronological backtracking that Prolog uses to implement non-determinism is itself a very inefficient strategy [Mel85]. However, Prolog predicates are typically written with one sole direction in mind and, as a result, some parameters are meant to be exclusively input or output. Knowledge of such directionality can be expressed using the notion of *modes*, a concept which was introduced by D.H.D. Warren (and refined by Mellish) to classify the ways in which a Prolog predicate is used during the execution of a program. If the programmer provides such clues to help the compiler identify directionality, the generated code can be drastically improved. A possible alternative is to infer the mode information by performing a global analysis of the program [DW88].

The most important use of mode information is as follows:

- determinacy analysis, which attempts to discover those predicates that are deterministic, with the potential benefit of reducing backtracking actions;

- special unification, that is, choosing specially-tailored routines to be used for those arguments whose type is known in advance;

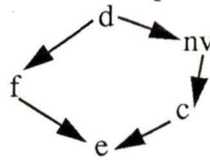
- clause indexing, which can be improved by narrowing the range of possible matching clauses given a particular mode for the arguments.

The standard approach for determining the mode information of a Prolog program statically uses *abstract interpretation* [CC77], [CC92]. This is a general technique where the standard semantics of a program are projected onto a different (and simpler) domain. Several solutions to the problem of finding the modes of a Prolog program have been proposed. A quite extensive survey is given in the introduction of [Deb89]. In this section, the mode inference algorithm of Debray [Deb89] is described since this framework is the basis of the determinacy analysis for the present thesis.

## 5.1 Modes

The mode of a predicate in a Prolog program specifies which arguments are input arguments and which are output arguments, taking into account all possible calls that can occur during the execution of a program. Depending on the nature of the problem, a set of modes must be defined to characterize the modes of the arguments in a Prolog predicate. Debray proposed the family of modes $\Delta$ = { **c**, **d**, **e**, **f**, **nv** }, where **c** denotes the set of fully-instantiated (ground) terms, **d** (don't know) the universal set of all terms, **e** the empty set, **f** (free variable) the set of un-instantiated variables, and **nv** the set of non-

variable terms (that is, structured terms which are not fully instantiated). The set $\Delta$ forms a complete lattice under the inclusion operator:




Given a set of terms $T$, its *instantiation* is defined to be the element of $\Delta$ that best characterizes it. Thus, the least upper bound [Bir40] for all terms in $T$ is chosen.

Prolog's unification operation can be understood in the mode's domain (called the *abstract* domain) as an operation that, given the instantiations of the arguments in a call, refines them according to the nature of the head arguments. Given two term instantiations $T_1$ and $T_2$, the unification of them is chosen to be the least upper bound of their instantiations under the following partial ordering:

$$\mathbf{f} \subseteq \mathbf{d} \subseteq \mathbf{nv} \subseteq \mathbf{c} \subseteq \mathbf{e}$$

The *join* operation of two elements of the lattice, $a$ and $b$, returns the least upper bound of $a$ and $b$. The join operator for the ordering under consideration is written as $\nabla$.

Some examples are shown in Figure 5.1. Note that, since some information is not taken into account in the abstract domain, the results are usually less accurate than in the concrete (and more complex) world.

## 5.2 General Mode Analysis Method

Debray's method uses the *procedural view* for Prolog, which considers the existence of mechanisms such as procedure call, success, failure, backtracking, etc. Debray's static inference of Prolog modes is based on keeping track of individual variable instantiations throughout the execution of a program. Such information is propagated in the usual way, from caller to callee at any predicate invocation and from callee to caller at the time of the predicate completion. Note that, with this method, the effects of variable aliasing are fully considered. Thus, at any point during program execution, an *instantiation state* is defined, which contains both instantiation and aliasing information for every variable in the program. Figure 5.2 contains an example of how this information is included within instantiation states for a given query. The instantiation information of a variable $v$ in an instantiation state A is referred to as *inst(A(v))*; the aliasing information is referred to as the *dependency set* of the state or *deps(A(v))*.

The notion of an instantiation state can be extended to any arbitrary non-variable term. A constant term will have ground instantiation (**c**) and an empty dependency set. A structured term will have ground instantiation (**c**) if all its arguments are ground and

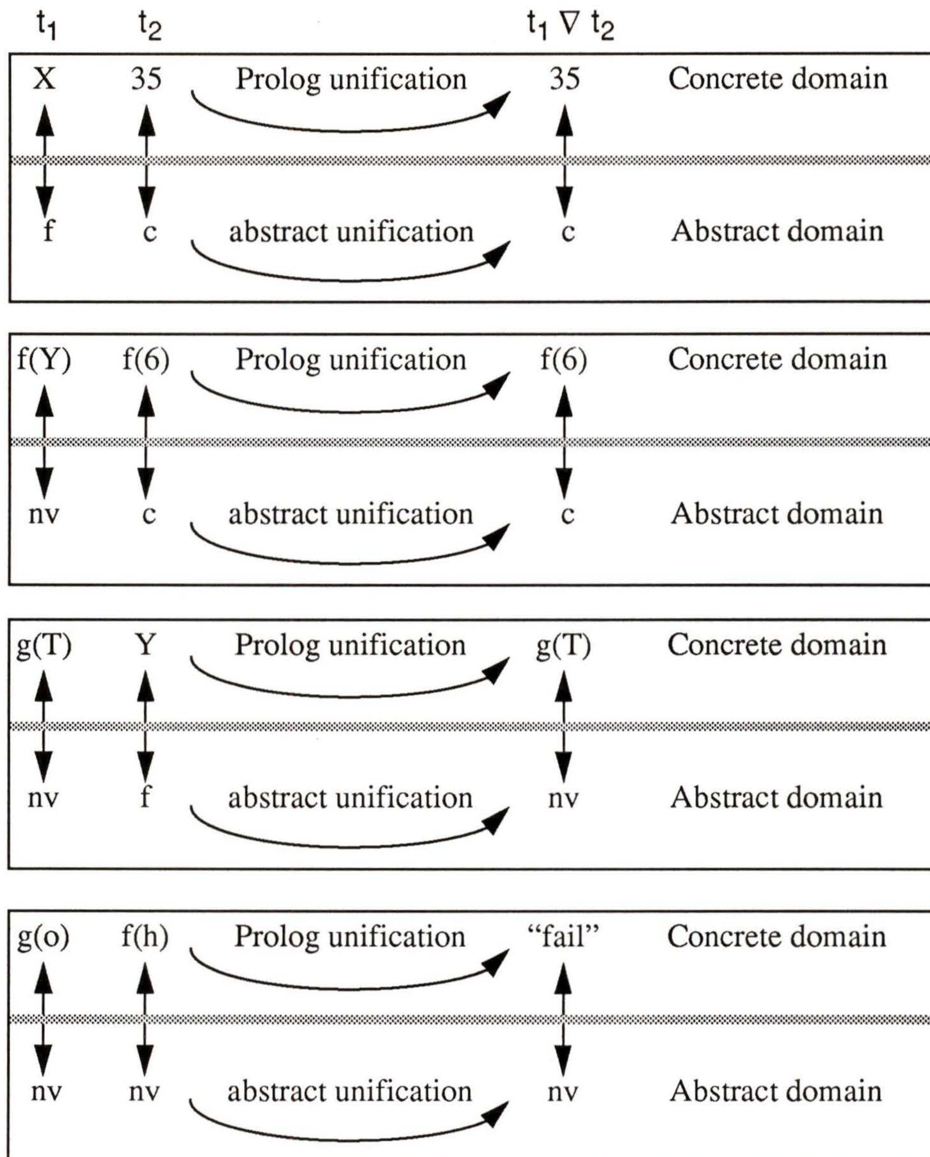



**Figure 5.1** Abstract interpretation applied to Prolog unification.

non-variable instantiation (**nv**) otherwise; its dependency set will be formed by the union of the individual dependency sets of its arguments.

In order to facilitate the propagation of mode information, Debray's method defines the existence of *instantiation patterns* for every procedure call. An instantiation pattern will contain, for every procedure argument, some information related to its instantiation and, additionally, an indication of those variables that are shared with other arguments. This is exemplified in Figure 5.3

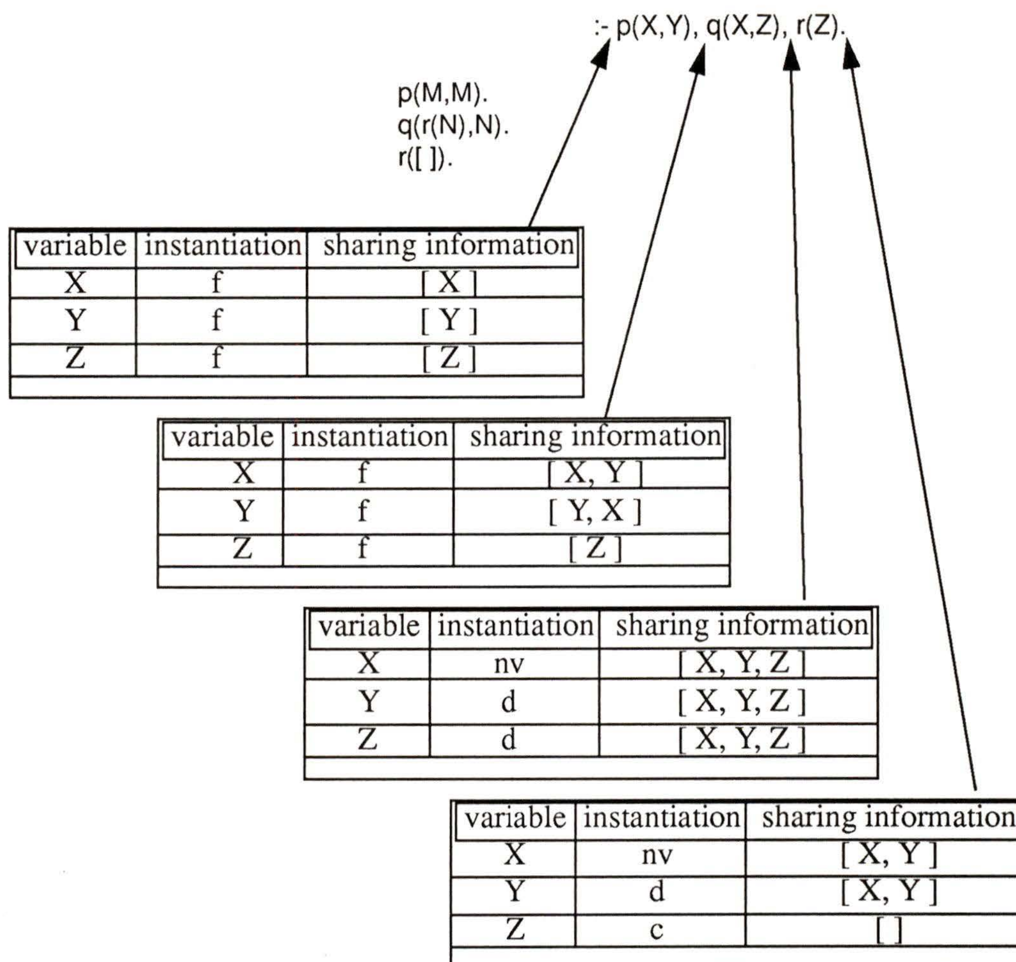



**Figure 5.2** Instantiation and sharing information in a goal.

### Abstract Unification

Using Debray's framework, Prolog's unification operation is performed on instantiation states in the abstract domain. After abstract unification, the instantiations and aliasing information for every variable are updated to reflect the effects of unifying the arguments of a procedure call with those in the head of a predicate clause.

The process of abstract unification can be separated into two processes: (a) the calculation of the new instantiations for all those variables involved in the procedure call, and (b) the determination of new variable dependencies as a result of the unification process.

To obtain the new instantiations for the variables that appear in the procedure call, it is necessary to determine the effects of a change of instantiation of an argument (that

:- ... , q(X,g(X,Z),Y,9), ...

| variable | instantiation | sharing information |
|---|---|---|
| X | f | [X,Y] |
| Y | f | [Y,X] |
| Z | f | [Z] |
| | | |

| argument | instantiation | variables and their dependencies | sharing information |
|---|---|---|---|
| 1 | f | [X] ⟶ <X,Y> | [1,2,3] |
| 2 | nv | [X,Z] ⟶ <X,Y>, <Z> | [1,2,3] |
| 3 | f | [Y] ⟶ <Y,X> | [1,2,3] |
| 4 | c | [ ] ⟶ <> | [4] |

**Figure 5.3** Instantiation patterns for a particular call.

is, a Prolog term) on the variables that occur in the term. The instantiations produced this way are called *inherited* instantiations. In general, a variable inherits the instantiation after unification that will be associated with the term in which it occurs. However there are two exceptions to this rule. The first exception takes into consideration the case when the subgoal's argument which contains the variable is unified to a free-variable (**f**) term: in this case, the subgoal's variable retains its previous instantiation. The other exception reflects the situation where a non-variable argument, as a result of unification, becomes instantiated to a non-variable (**nv**) term: if the subgoal term that contains the variable under consideration has a non-variable (**nv**) instantiation prior to the abstract unification, its inherited instantiation will be a don't know (**d**) instantiation because it is not possible to determine precisely whether the variable will be affected as a consequence of the concrete unification (there is not enough information in the abstract domain to give an accurate answer). Some common combinations are listed in Figure 5.4.

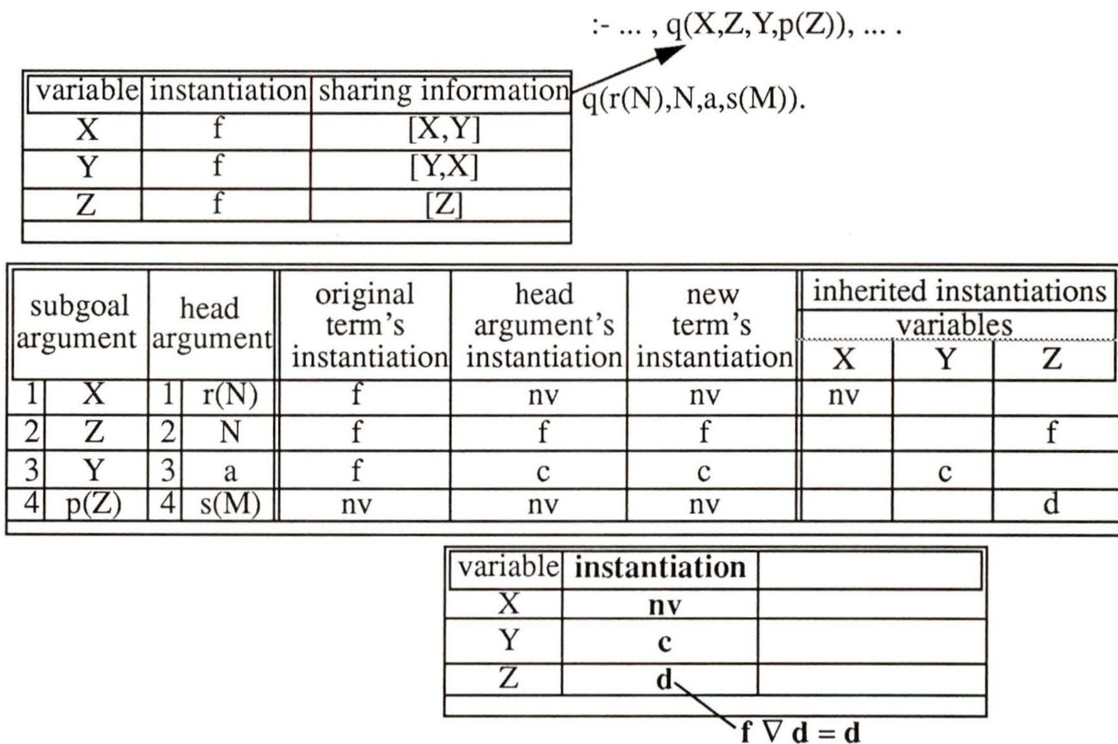
Once the inherited instantiations have been found, the individual instantiation for every variable is chosen to be the least upper bound of all the possible inherited instantiations that apply to that variable (according to the partial order introduced earlier). An example of this appears in Figure 5.5.

The new variable dependencies are calculated by analyzing those arguments in which a given variable appears. By applying transitive closure to the instantiation pattern that describes the procedure call, those arguments that share variables with the argument containing the variable under analysis are obtained. The new variable dependencies are constructed by concatenating all the discrete variable dependencies associated with each of those terms. This is illustrated in Figure 5.6.

| subgoal term | head argument | X variable's old instantiation | term's old instantiation | term's new instantiation | X variable's new instantiation |
|---|---|---|---|---|---|
| f(X,a,R) | f(N,a,T) | f | nv | nv | f |
| f(X,a,R) | f(a,N,T) | f | nv | nv | c |
| f(X,a,R) | f(N,a,T) | nv | nv | nv | nv |
| f(X,a,R) | f(a,N,T) | nv | nv | nv | c |
| f(X,a,R) | f(N,a,T) | d | nv | nv | d |
| f(X,a,R) | f(a,N,T) | d | nv | nv | c |
| f(X,a,R) | f(a,N,T) | e | nv | nv | e |

Note that when a non-variable subgoal argument remains non-variable after unification nothing can be said about the instantiation of the variables that occur in it. For this reason, its 'inherited' instantiation is set to *don't know* (**d**).

**Figure 5.4** Inherited instantiations for some term instantiations.



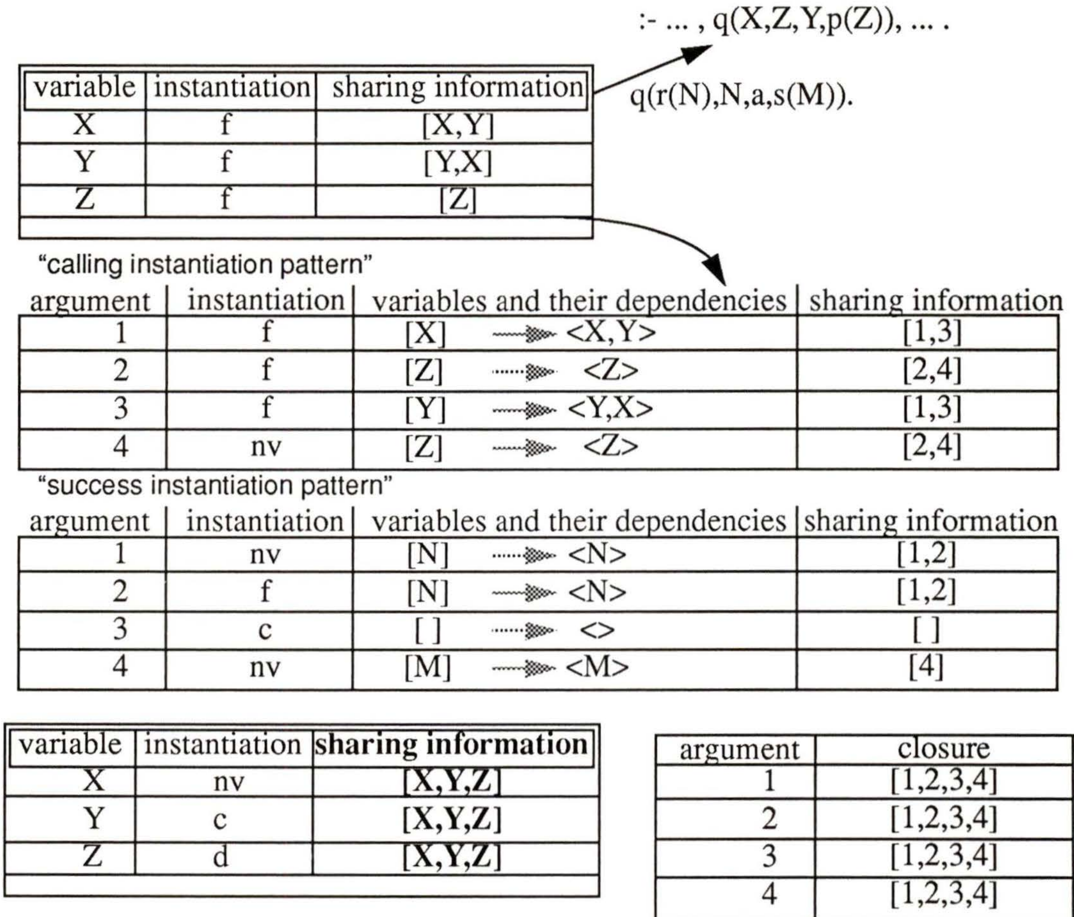

| variable | instantiation | sharing information |
|---|---|---|
| X | f | [X,Y] |
| Y | f | [Y,X] |
| Z | f | [Z] |

| subgoal argument | | head argument | | original term's instantiation | head argument's instantiation | new term's instantiation | inherited instantiations variables X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|
| 1 | X | 1 | r(N) | f | nv | nv | nv | | |
| 2 | Z | 2 | N | f | f | f | | | f |
| 3 | Y | 3 | a | f | c | c | | c | |
| 4 | p(Z) | 4 | s(M) | nv | nv | nv | | | d |

| variable | **instantiation** | |
|---|---|---|
| X | **nv** | |
| Y | **c** | |
| Z | **d** | |

**Figure 5.5** Variable instantiations after unification.

The next step is to propagate both the changes of instantiations and aliasing information that are the result of dependencies between variables. The propagation of aliasing information is straightforward because the new information is just added to those variables that depend on the variables which suffered changes after unification. The



:- ... , q(X,Z,Y,p(Z)), ... .

q(r(N),N,a,s(M)).

| variable | instantiation | sharing information |
|---|---|---|
| X | f | [X,Y] |
| Y | f | [Y,X] |
| Z | f | [Z] |

"calling instantiation pattern"

| argument | instantiation | variables and their dependencies | sharing information |
|---|---|---|---|
| 1 | f | [X] → <X,Y> | [1,3] |
| 2 | f | [Z] → <Z> | [2,4] |
| 3 | f | [Y] → <Y,X> | [1,3] |
| 4 | nv | [Z] → <Z> | [2,4] |

"success instantiation pattern"

| argument | instantiation | variables and their dependencies | sharing information |
|---|---|---|---|
| 1 | nv | [N] → <N> | [1,2] |
| 2 | f | [N] → <N> | [1,2] |
| 3 | c | [ ] → <> | [ ] |
| 4 | nv | [M] → <M> | [4] |

| variable | instantiation | **sharing information** |
|---|---|---|
| X | nv | **[X,Y,Z]** |
| Y | c | **[X,Y,Z]** |
| Z | d | **[X,Y,Z]** |

| argument | closure |
|---|---|
| 1 | [1,2,3,4] |
| 2 | [1,2,3,4] |
| 3 | [1,2,3,4] |
| 4 | [1,2,3,4] |

**Figure 5.6** Aliasing information of variables after unification.

propagation of changes of instantiation is only relevant when the variable under consideration is uninstantiated and the instantiation of one of its possible aliases is modified.

A final step consists of "cleaning-up" the aliasing information of those variables that have changed their instantiation to "ground", in which case the aliasing information must be reset (this is because a ground term does not share variables with another variable).

# Chapter 6. ECTOPIC, an Improved Translator

The code generated by the TOPIC is inferior to WAM code in at least two major ways. Firstly, last call optimization (and a special case, tail recursion) is not implemented. Secondly, the TOPIC system does not apply clause indexing techniques. As a result of the omission of those two important WAM optimizations, the code generated by the TOPIC system is not efficient for many common Prolog programs. In fact, the absence of last call optimization may prevent a Prolog program from being able to run at all.†

The addition of optimization techniques to the code generated by the TOPIC system has instigated a new implementation of the translator from Prolog into C++. The C++ output is compatible with the TOPIC design. This new translator, named ECTOPIC, which is the subject of the present thesis, has been entirely written in Prolog and incorporates tail recursion optimization and clause indexing in the generated code.

This chapter is organized as follows. An introductory section provides a justification for writing the translator in Prolog. The rest of the chapter discusses the implementation of tail recursion optimization and clause indexing in the TOPIC system environment.

## 6.1 A Translator Written in Prolog

Once the decision of adding suitable optimizations to the TOPIC system was reached, the issue of how to achieve such improvements had to be considered. It was decided to write a new translator in a different programming language from the C language originally used for the TOPIC system. Two circumstances biased the selection in favour of the use of Prolog: the ease of implementation of a parser for Prolog and the relatively straightforward implementation of mode analysis using Prolog.

First, writing a Prolog parser in the Prolog language is surprisingly easy. With the help of built-in predicates, clause parsing is trivial. For example, Prolog provides predicates to read entire clauses, and to determine the type of every term or to establish the arity of a functor. Even some more difficult semantic aspects, such as operator precedence, can easily be managed using Prolog built-in predicates.

Secondly, given the nature of mode analysis, which is just another fixed-point problem, the Prolog language seems to be a natural candidate for programming this kind of algorithm. In fact, Prolog implementations to find fixed-points appear in the literature; see for example [OKe87].

---

† Recall that last call optimization re-uses part of the stack when the last goal of a clause is invoked. The absence of this optimization may result in an unwanted growth of the stack.

## 6.2 Adding Prolog Optimizations to the Translator

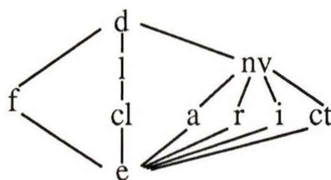
Two major Prolog optimizations were added to the translator: tail recursion optimization and clause indexing. Since these optimization techniques have been discussed in a previous chapter, we show only how the mode analysis framework can be used to perform the optimizations.

To implement tail recursion optimization, it is necessary to find out which of the program predicates are deterministic, since they are the ideal candidates for this kind of optimization. The determination of deterministic predicates requires both mode analysis and knowledge of the places where the different cut predicates appear in the distinct clauses of a Prolog predicate. The first sections of this chapter are devoted to the improvement of the TOPIC code via determinacy and mode analysis.

Clause indexing can be implemented in a straightforward way using the first argument of every Prolog clause consistently. It is worth pointing out that some predicates that are normally not deterministic may become deterministic by combining mode analysis and clause indexing. The concluding sections of this chapter cover some topics related to the addition of clause indexing to the TOPIC system.

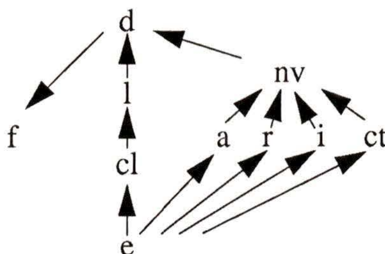
### 6.2.1 Mode Analysis

This initial section sketches an implementation of mode analysis for Prolog programs. This particular implementation is based on the mode analysis framework that was described in Chapter 5. However, the TOPIC system provides a richer set of Prolog types. Debray's original lattice has been extended to distinguish lists, integers and reals:




The set of modes is = [**ct**, **a**, **r**, **d**, **e**, **f**, **nv**, **l**, **cl** ], where **ct** (constant term) denotes the set of fully-instantiated structures, **a** (atom) the set of atomic terms, **i** (integer) the set of integer numbers, **r** (real) the set of real numbers, **d** (don't know) the universal set of all terms, **e** the empty set, **f** (free variable) the set of un-instantiated variables, **cl** (constant list) the set of fully-instantiated lists, **l** (list) the set of Prolog lists, and **nv** the set of non-variable (non-list) terms. For Prolog unification purposes, the unification of two

term instantiations $T_1$ and $T_2$ is chosen to be the least upper bound of their instantiations under the following partial ordering:




The main advantage of using a more complex lattice is that the results of mode analysis reflect the TOPIC class hierarchy for Prolog terms more directly. If an argument is known to have a particular mode, more specific TOPIC routines can be generated. Unfortunately, the addition of extra modes requires changes to Debray's framework, with the disadvantage of making the algorithms more complex since more cases must be taken into account.

The Prolog implementation of mode analysis consists of two main groups of modules. A first group of routines performs the actualization of the *instantiation states* that characterize the different program points that can be reached when the Prolog program is executed. The previously described framework of Debray is used to update the mode of the arguments as a result of the process of head unification. A second group of routines propagates the flow information taking into account all possible program execution paths. Given initial information (provided by the user) about the modes of the arguments in a call to the *external* predicates, i.e., those predicates that can be called from outside the Prolog program, a dataflow analysis of the program is launched. The *calling patterns* provided by the user are used to obtain the modes of the arguments in a call to every predicate that is invoked during the program execution. Since more than one set of modes is possible for the arguments of a given predicate, an iterative process computes new modes for the arguments successively until no more new modes are found, in other words, until a fixed point has been found for the dataflow problem.

No major changes to Debray's framework were required. The only significant changes were concerned with the new lattice, which only required taking more cases into account.

### 6.2.2 Determinacy Analysis

Once the modes of the arguments of a given predicate have been calculated, this information is combined with the presence of Prolog cuts in the clauses of the predicate to establish if the predicate is determinate. In general, a predicate is determinate if [MEL85]:

- each clause apart from the last includes a "cut" as a conjunct,
- each predicate which does not occur before a "cut" in one of the clauses is itself determinate.

In an arbitrary Prolog program, there exists an interdependency amongst all predicates. It is possible to convert the problem into a *dataflow* problem. The naïve method for solving it is reduced to the problem of finding the *least fixed point* [OKe87] of the equations that describe the problem.

If mode information is available, the results of determinacy analysis can be more accurate. If some arguments are known to be ground (i.e., constant lists, atoms, integer numbers, real numbers, or constant terms), the analysis can be restricted to only those clauses that are unifiable to the particular ground mode. For example, for the predicate in Figure 6.1, suppose that it has been determined that the first argument is always instantiated to an integer before head unification takes place. Under this assumption, the determinacy analysis can be confined to only those clauses that have a first argument that is a variable or an integer. Thus, according to the above-mentioned rule, predicate $p$ is determinate if predicates $r$ and $s$ are determinate. Note that without mode information, $p$ cannot be proven to be determinate.

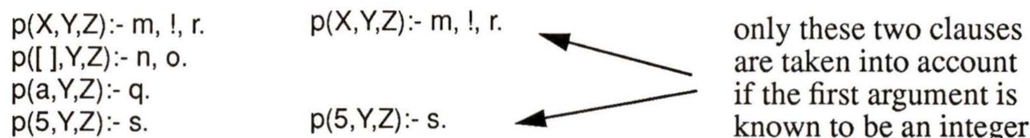



**Figure 6.1** Mode analysis can improve the results of determinacy analysis.

### 6.2.3 Tail Recursion Optimization

One key optimization that can be incorporated to the TOPIC system is tail recursion optimization (TRO). TRO can be performed when a predicate is determinate and the last subgoal (in the last predicate clause) is a recursive call to the same predicate. The optimization permits the recursive call to be treated as a simple iteration process. In other words, focusing on the TOPIC system, instead of generating code for the allocation of a new instance of the predicate, a simple jump to the beginning of the predicate's *Next* method (which we know only produces one answer) suffices. Some minor changes are necessary, namely the utilization of *LocalVariable* instances for the predicate arguments, and a scheme that permits a systematic allocation and deallocation of those local variables. We now proceed to explain these modifications in more detail.

The TOPIC system normally translates a final recursive call using the standard translation pattern, as follows:

- creation of a new *Predicate* instance for the last subgoal (the different arguments of the predicate, represented by data fields in the predicate instance, are assigned the initial values as specified in the subgoal);
- explicit invocation of the *Next* method for this new instance.

If we want to perform TRO, the TOPIC translation scheme is modified as follows:

- allocation of a stack frame to contain the predicate's variables;
- direct argument assignment according to the values specified in the last subgoal (no new instance is created; the argument values are only updated);
- insertion of a jump instruction to the beginning of the predicate's *Next* method.

However these are not the only modifications that are required. Unification functions must make references to the local variables allocated on the stack instead of the predicate instance's variables. Also, all the frames that have been allocated to accommodate TRO must be deallocated after the *Next* method gives its only answer or fails. This implies the necessity of keeping track of the number of times that a new frame of local variables has been allocated to ensure that an identical number of deallocations will take place once the *Next* method has completed its execution. All this is illustrated in Figure 6.2.

### 6.2.4 Clause Indexing

Another typical Prolog optimization is clause indexing. Normally, clause indexing is performed on the first argument of a given predicate. In the TOPIC system, the scheme for clause indexing is more complicated than in the WAM, inasmuch as TOPIC has no direct control over the program counter.

To implement clause indexing in the TOPIC system, two different issues must be considered. The first issue is related to control flow, that is, how to force, for a given argument type and position, the execution of only those clauses that are unifiable. The second issue has to do with the correct application of the TOPIC backtracking functions. Consider the example in Figure 6.3. Suppose that the first argument is always instantiated to a constant list before head unification. For such a case, only those clauses whose first argument is either a variable or a list should be tried. Since only two such clauses comply with that requirement, a *Checkpoint* instruction must precede the translation for the first clause and a *Restore* instruction must precede the second of the clauses. Now suppose that the first argument is always instantiated to a Prolog atom. Again,

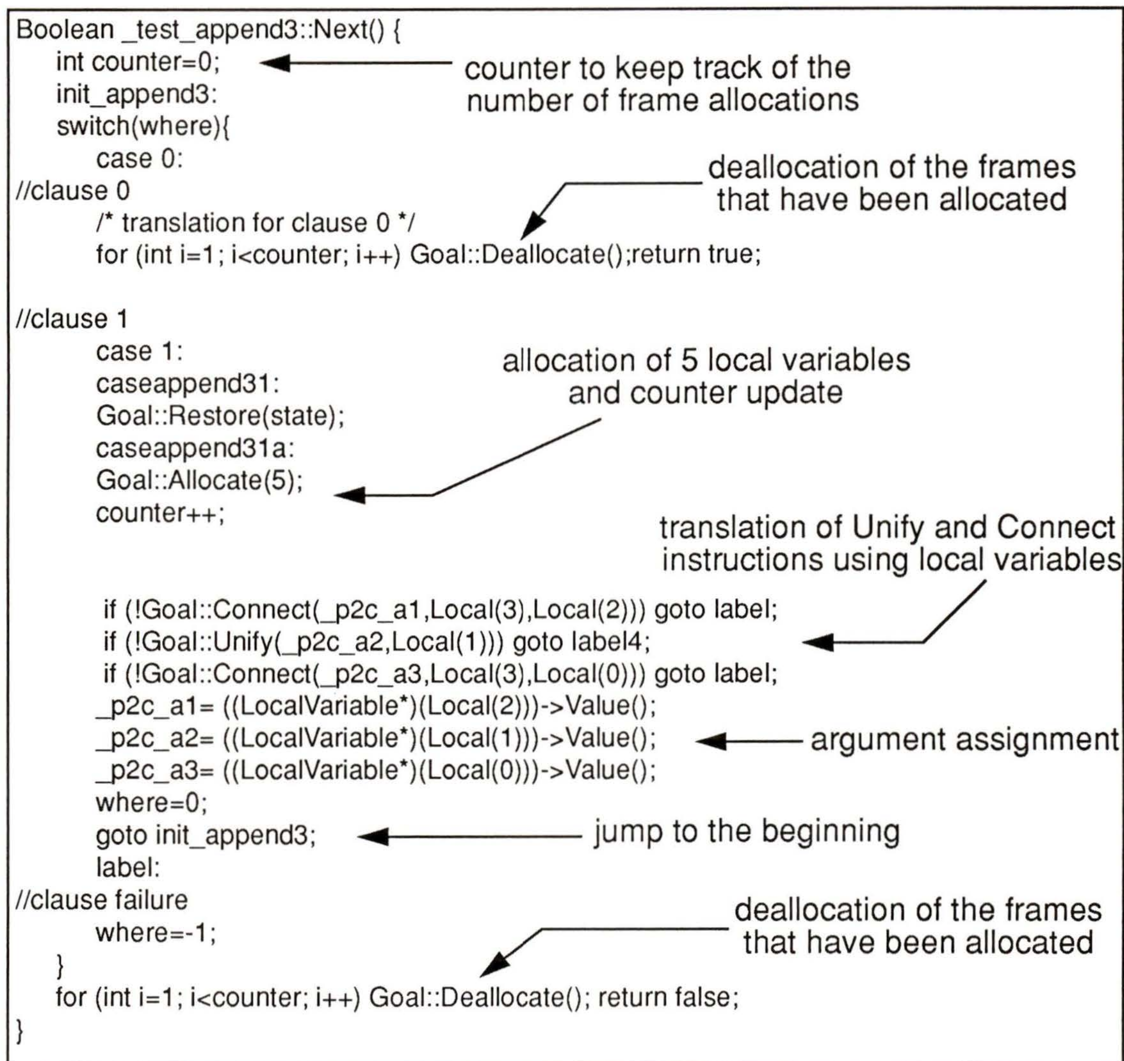



**Figure 6.2** Tail recursion optimization in the TOPIC system.

two clauses are to be explored, both preceded by the *Checkpoint* and *Restore* instructions, respectively. The reader should notice that the translation for the clause with a first argument variable (i.e., the fourth clause in our example) is preceded by a *Restore* instruction for the constant list case, whereas it is preceded by a *Checkpoint* instruction in the other case.

The implementation of clause indexing requires identification of which Prolog types are unifiable with a given argument position for every clause in a predicate. If, for every Prolog type and, within a given type, for every functor or constant value, we build

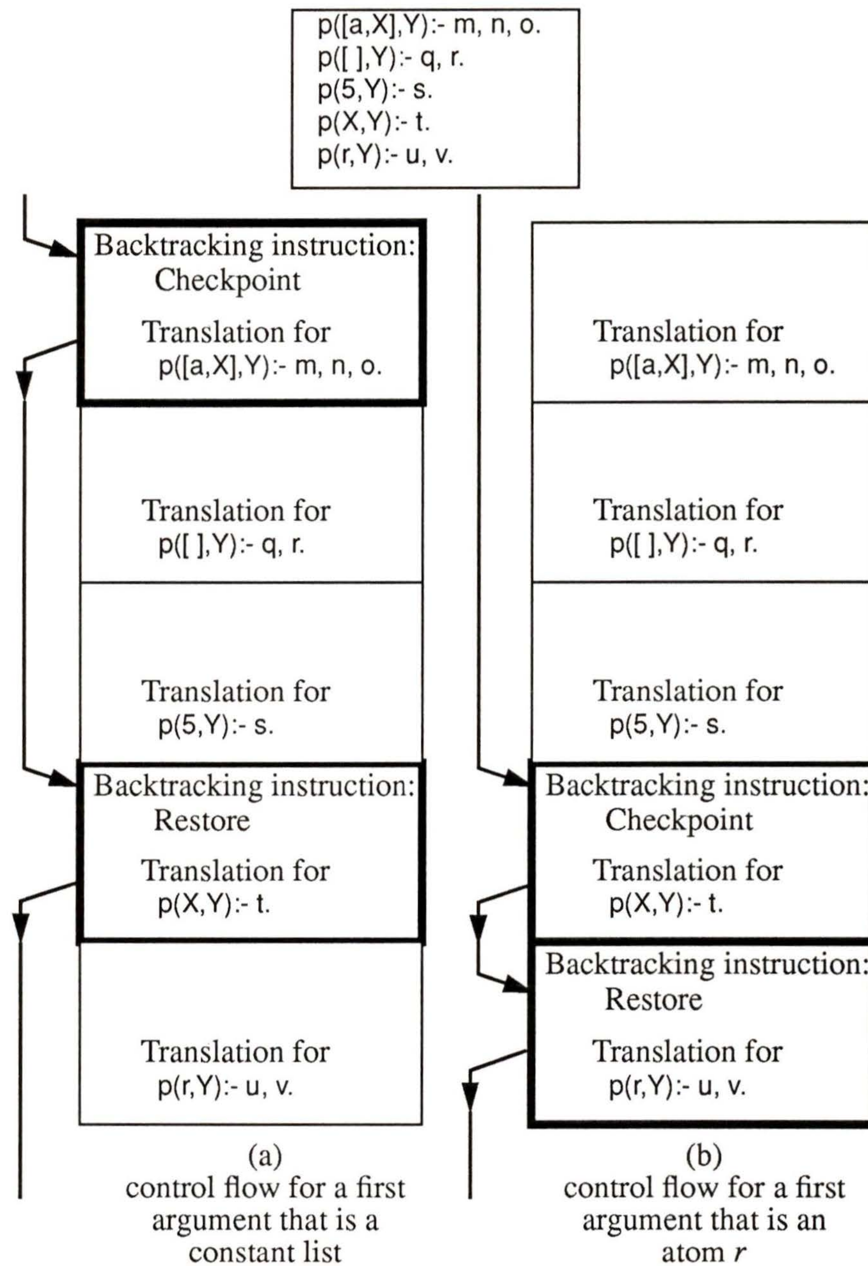



**Figure 6.3** Control flow for clause indexing.

a queue (which we name the *compatible clause queue*) containing the (numbers of the) clauses that are unifiable to the given argument position, the problem of clause indexing can be reduced to the problem of determining the following for each predicate clause:

- the argument types for the given position that can reach the particular clause which we call the *unifiable types*;

- for every unifiable type, both the relative position of the particular clause within the compatible clause queue for that particular Prolog type, and the next (if any) clause in the queue must be determined.

At the beginning of a predicate's *Next* method, a group of instructions determine the initial clause to be executed given the type (and functor or constant value) of the argument position under consideration. This initial clause is naturally given by the first element of the corresponding compatible clause queue. At the end of the code for every clause, another group of instructions decides the next clause to be executed for the argument type (and functor or constant value). This clause is just the next clause as indicated in the compatible clause queue. To resolve what kind of backtracking instruction must be used, the relative position of the clause in the compatible clause queue is considered as follows: (a) if the compatible clause queue only contains one (or zero) clause(s), no backtracking instructions need to be used; (b) otherwise, the first clause should be preceded by a *Checkpoint* instruction, the last clause must be preceded by a *Restore* instruction, and the rest (if any) need to start with a *Backtrack* instruction. All these backtracking operations must be handled separately for each Prolog type.

# Chapter 7. Experimental Results and Conclusions

In this chapter, we compare the efficiency between the code generated by our translator and the code generated by the TOPIC system. This comparison gives special consideration to programs where tail recursion optimization and clause indexing are applicable, since in other cases outputs from the two systems are virtually the same. At the end of the chapter we summarize some conclusions and suggest directions for further work.

## 7.1 Benchmarks

ECTOPIC is able to generate better code for some programs with a tail-recursive nature. Table 7.1 compares the performance of our translator with the TOPIC system for the naïve reverse example (cf. Figure 7.1) for different sizes of the list to be inverted (the naïve reverse program is a typical benchmark for testing recursive programs). Note that in this example, the amount of stack memory grows linearly with the input list length in the code produced by our translator, whereas the growth is quadratic in the TOPIC system case.

```
nrev( [ ], [ ] ).
nrev( [X|Y], Z ) :-
    nrev(Y,YR), append( YR, [X], Z ).

append( [ ], X, X ).
append( [X|XL], Y, [X|ZL] ) :-
    append( XL, Y, ZL ).

test(N) :- makelist(N,L), nrev(L,R), write(R).

makelist(0,[ ]) :- !.
makelist(N,[N|R]) :- M is N-1, !, makelist(M,R).
```

**Figure 7.1** The naïve reverse program.

Another group of programs for which ECTOPIC produces a better code (at least under the point of view of stack management) consists of those programs that do not require the allocation of backtracking information as a result of having incompatible first arguments in the heads of the clauses (if such an argument is known to be ground). A typical program that shows this characteristic is the quicksort example in Figure 7.2. Since the TOPIC system does not distinguish the type of the first arguments in the heads of the clauses, it systematically allocates backtracking information on the stack, information that is not removed until the execution of the next subgoal, a situation that can result in undesirable stack growth.

| list length | ECTOPIC | TOPIC |
|---|---|---|
| 0 | 0 | 24 |
| 5 | 780 | 1904 |
| 10 | 1560 | 5984 |
| 15 | 2340 | 12264 |
| 20 | 3120 | 20744 |
| 25 | 3900 | 31364 |
| 30 | 4680 | 44304 |
| 35 | 5460 | 59384 |
| 40 | 6240 | - |
| 45 | 7020 | - |
| 48 | 7488 | - |
| ... | ... | ... |
| N | $156n$ | $44n^2 + 156n + 24$ |

(stack growth in bytes)

Note: predicate test(N) fails in TOPIC for N>36 (message: failed - stack exhausted), and, in our translator, for N>48 (message: failed - heap exhausted). Note that in the latter case the failure is not due to a bad implementation of TRO but to memory constraints in the space reserved to the creation of new Prolog terms).

**Table 7.1** TRO benchmark.

The final family of Prolog programs whose ECTOPIC code is more efficient with respect to the TOPIC code consists of database-like programs. Table 7.2 gives the corresponding comparison for a program especially suited for clause indexing, a simple path finder for a directed graph; Table 7.3 shows the timings for a query on a population and area database of several countries to find countries of approximately equal population density. All times are user times measured on a Sun SPARCstation† SLC. The source programs are listed in Appendix 1. The test programs were measured several times on an unloaded system; the minimum time was taken. It is not surprising that

†SPARCstation is a trademark of Sun Microsystems, Inc.

```
% David H. D. Warren
%
% quicksort a list of 50 integers

qsort :- qsort([27,74,17,33,94,18,46,83,65, 2,
    32,53,28,85,99,47,28,82, 6,11,
    55,29,39,81,90,37,10, 0,66,51,
     7,21,85,27,31,63,75, 4,95,99,
    11,28,61,74,18,92,40,53,59, 8],_,[ ]).

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

partition([X|L],Y,[X|L1],L2) :-
    X =< Y, !,
partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) :-
partition(L,Y,L1,L2).
partition([ ],_,[ ],[ ]).
```

In the TOPIC system, the execution *in tandem* of two or more invocations to *qsort* fails as a result of not enough stack memory. In ECTOPIC, the user can concatenate as many *qsort* subgoals as desired (with the only limitation of heap space).

**Figure 7.2** The quicksort example.

clause indexing optimization speeds up the execution of the testing program with respect to the program without clause indexing.

## 7.2 Conclusions and Future Work

Mode and determinacy analysis of Prolog programs provide essential information to standard optimization techniques such as tail recursion optimization and clause indexing. The importance of tail recursion optimization is particularly evident when more economical use of memory can make the difference between a program that is able to run and a program that is not. On the other hand, clause indexing is useful for predicates that have a large number of clauses with disjunct (ground) arguments. However, clause indexing can be useless when the argument position that is used for performing the clause selection has no significantly different ground values to discriminate between,

| number of searches | TOPIC | ECTOPIC | ratio |
|---|---|---|---|
| 5 | 0.35 | 0.18 | 1.94 |
| 10 | 0.66 | 0.34 | 1.94 |
| 15 | 0.94 | 0.49 | 1.91 |
| 20 | 1.23 | 0.62 | 1.98 |
| 25 | 1.51 | 0.79 | 1.91 |
| 30 | 1.76 | 0.94 | 1.87 |

User time (seconds).

**Table 7.2** Path finder program benchmark.

| number of invocations | TOPIC | ECTOPIC | ratio |
|---|---|---|---|
| 1 | 0.82 | 0.20 | 4.10 |
| 2 | 1.52 | 0.33 | 4.60 |
| 3 | 2.26 | 0.53 | 4.26 |
| 4 | 3.01 | 0.68 | 4.42 |
| 5 | 4.50 | 0.97 | 4.63 |
| 6 | 5.39 | 1.15 | 4.68 |

User time (seconds).

**Table 7.3** Query example benchmark.

or when the number of clauses is small. Additional work would include improving the clause indexing scheme in such a way that it is guaranteed to be used only when a major gain is obtainable.

In its implemented form, determinacy analysis is only performed on conjunctive ("and") subgoals. Therefore, other kinds of subgoals are not considered yet, as for example, disjunctive ("or") and conditional ("guard") subgoals. To deduce general rules for these subgoals would require finding transformations that generate equivalent con-

junctive subgoals while preserving soundness. A more difficult task (and very likely, an expensive one) would be to try to extend the analysis to dynamic code (thus, allowing the existence of predicates such as *assert* or *retract*). In fact, this analysis would be restricted to identify those parts of a Prolog program that are independent of (i.e., unaffected by) the run-time changes induced by the above-mentioned predicates. Those parts could be analyzed and optimized using the standard techniques [DW88].

Another optimization that can be added to ECTOPIC is the inclusion of specially-tailored unification routines. If we know in advance the types of the arguments before unification, the utilization of unification routines that assume the type of the arguments that are involved can represent time savings.

Similarly, it is possible to develop a better (i.e., more intelligent) clause indexing scheme. In fact, complete indexing techniques have been proposed in the literature [Han92]. However, if a simpler scheme is preferred, some modest improvements can be applied. Firstly, indexing can be extended to arguments other than the first one. Secondly, it is not always appropriate to perform clause indexing (for example if the number of clauses is too small), thus a scheme that determines when it is worthwhile to perform clause indexing can be devised. Finally, in the case of arguments that are structures, additional indexing over the first argument of the structure can be incorporated (a fourth level indexing).

The ECTOPIC system does not perform an analysis to determine if the clauses within a predicate are mutually exclusive. As a result, some determinate predicates may not be discovered, unless the user provides some clue (in the form of the insertion of cuts) or the user restructures the code. Therefore, this kind of analysis could be implemented to find more determinate predicates to which the standard optimizations can be applied.

Finally, additional work can be directed to the simplification of some C++ routines, when we are dealing with determinate predicates. Since determinate predicates do not require backtracking information, the corresponding *Next* method could be implemented via a normal (and simpler) C function that returns the only solution.

# Bibliography


[Ait91] Aït-Kaci, H. *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, Mass., 1991.

[AC72] Allen, F.E., and Cocke, J. *A Catalogue of Optimizing Transformations*, in *Design and Optimization of Compilers*, Randall, R. [editor], Prentice-Hall, 1972.

[Bir40] Birkhoff, G. *Lattice theory*. American Mathematical Society Colloquium Publications, Vol. 25, New York, 1940.

[BDMN73] Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B, and Nygaard, K. *SIMULA Begin*. New York: Petrocelli/Charter, 1973.

[Bra86] Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, Wokingham, England, 1986.

[Byr80] Byrd, L. Understanding the Control Flow of Prolog Programs. *Proceedings of Logic Programming Workshop*. Debrecen, Hungary, 1980.

[CM81] Clocksin, W.F., and Mellish, C.S. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[CC77] Cousot, P., and Cousot, R. Abstract Interpretation: a Unified Framework for Static Analysis of Programs by Construction of Approximation of Fixpoints. *Proceedings of the 4th ACM Conference on Principles of Programming Languages*, The Association for Computing Machinery, New York, N.Y., 1977, pp. 238-252.

[CC92] Cousot, P., and Cousot, R. *Abstract Interpretation and Application to Logic Programs*. Laboratoire d'Informatique de l'École Normale Supérieure, Research Report LIENS-92-12, June 1992.

[DW88] Debray, S.K. and Warren, D.S. Automatic Mode Inference for Logic Programs. *The Journal of Logic Programming*, Vol. 5 no. 3, Sept. 1988, pp. 207-229.

[Deb89] Debray, S.K. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, Vol. 11 No. 3, July 1989, pp. 418-450.

[ES90] Ellis, M.A., and Stroustrup, B. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Mass., 1990.

[Hai86] Hailpern, B. Multiparadigm Languages and Environments. *IEEE Software*, Vol. 3, No. 1, Jan 1986, pp. 6-9.

[Han92] Hans, W. *A Complete Indexing Scheme for WAM-based Abstract Machines*. RWTH Aachen, Lehrstuhl für Informatik II, Research Report 92-11, 1992.

[HMS88] Hayes, R., Manweiler, S.W., and Schlichting, R.D. A Simple System for Constructing Distributed, Mixed-Language Programs. *Software — Practice and Experience*, Vol. 18, No. 7, Jul 88, pp. 641-660.

[HN87] Hailpern, B., and Nguyen, V. A Model for Object-Oriented Programming, in *Research Directions in Object-Oriented Programming*, edited by Striver, B., and Wegner, P., MIT Press, 1987.

[Hug91] Hughes, D.K. Multilingual Software Engineering Using Ada and C. *ACM SIGSOFT Software Engineering Notes*, Vol. 16, No. 4, pp. 55-59.

[Hor51] Horn, A. On Sentences which are True of Direct Unions of Algebras. *Journal of Symbolic Logic*, Vol. 16, pp. 14-21. Referenced in [Kow79].

[Jor90] Jordan, D. Implementation Benefits of C++ Language Mechanisms. *Communications of the ACM*. Vol. 33, No. 9, September 1990, pp. 61-64.

[JLH90] Junkin, M.D., Levy, M.R., and Horspool, R.N. *The Translation of Prolog into C++*. University of Victoria, British Columbia, Canada, 1990.

[Jun90a] Junkin, M.D. *The TOPIC Translator Front End.* University of Victoria, British Columbia, Canada, 1990.

[Jun90b] Junkin, M.D. *The TOPIC Translator Back End.* University of Victoria, British Columbia, Canada, 1990.

[Jun90c] Junkin, M.D. *The Internal Design of the TOPIC Class Library*. University of Victoria, British Columbia, Canada, 1990.

[Jun90d] Junkin, M.D. *Using the TOPIC System*. University of Victoria, British Columbia, Canada, 1990.

[Kow74] Kowalski, R. Predicate Logic as Programming Language. *Proceedings of IFIP-74*, North-Holland, 1974, pp. 569-574.

[Kow79] Kowalski, R. *Logic for Problem Solving*, Elsevier computer science library, New York: North-Holland, 1979.

[Lie86] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *OOPSLA '86 Conference Proceedings*. The Association for Computing Machinery, edited by N. Meyrowitz, New York, N.Y., 1986, pp. 214-223.

[Mel85] Mellish, C.S. Some Global Optimizations for a Prolog Compiler. *The Journal of Logic Programming*, vol. 2 no. 1, April 1985, pp. 43-66.

[OKe87] O'Keefe, R.A. Finite Fixed-Point Problems. *Logic Programming*. Proceedings of the Fourth International Conference (edited by J.-L. Lassez), Vol. 2, pp. 729-743.

[Prz90] Przymusinski, T.C. *Non-monotonic Reasoning versus Logic Programming: a New Perspective*, in *The Foundations of Artificial Intelligence*, edited by Partridge, D., and Wilks, Yorick, Cambridge University Press, Cambridge, UK, 1990.

[Sha84] Shaw, M. Abstraction Techniques in Modern Programming Languages. *IEEE Software*. Vol. 1, No. 4, October 1984, pp 10-27.

[SS87] Sterling, L. and Shapiro, E. *The Art of Prolog*. The MIT Press, Cambridge, Mass., 1987.

[Str86] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 1986.

[Str88] Stroustrup, B. What is Object-Oriented Programming? *IEEE Software*. Vol. 5, No. 3, May 1988, pp. 10-20.

[Tic88] Tick, E. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers. Norwell, Mass., 1988.

[War77] Warren, D.H.D. *Implementing Prolog —Compiling Predicate Logic Programs*. Research reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[War80] Warren, D.H.D. *An Improved Prolog Implementation which Optimises Tail Recursion*. Research Paper 156, Department of Artificial Intelligence, University of Edinburgh, 1980. A brief overview can be found in Tick's book.

[War83] Warren, D.H.D. *An Abstract Prolog Instruction Set*, Technical Report # 309, SRI International, 1983.

[Wat90] Watt, D.A. *Programming Language Concepts and Paradigms*. Prentice-Hall, New York, N.Y., 1990.

[Weg84] Wegner, P. Capital Intensive Software Technology. *IEEE Software*, Vol. 1, No. 3, July 1984, pp. 7-45.

[Weg86] Wegner, P. *Perspectives on Object-Oriented Programming*. Department of Computer Science, Brown University, Technical Report No. CS-86-25, December 1986.

[Wil91] Wileden, J.C. *et al.* Specification-Level Interoperability. *Communications of the ACM*, Vol. 34, No. 5, May 91, pp. 73-87.

[Wol89] Wolfe, W. A Practical Comparison of Two Object-Oriented Languages. *IEEE Software*. Vol. 6, No. 5, September, 1989, pp. 61-69.

# Appendix 1 Test Programs

## Path Finder Program

```
% path finder
path(A,C):-edge(A,B),edge(B,C).
path(A,C):-path(A,B),edge(B,C).
edge(a,a).
edge(a,b).
edge(a,c).
edge(a,d).
edge(a,e).
edge(a,f).
edge(a,g).
edge(b,a).
edge(b,b).
edge(b,c).
...
(47 lines omitted)
...
edge(h,i).
edge(i,a).
edge(i,b).
edge(i,c).
edge(i,d).
edge(i,e).
edge(i,f).
edge(i,g).
edge(i,h).
edge(i,i).
```

## Query Program

```
% query
%
% David H. D. Warren
%
% query population and area database
%to find countries of approximately equal
%population density

query :- query(_), fail.
query.

query([C1,D1,C2,D2]) :-
    density(C1,D1),
    density(C2,D2),
    D1 > D2,
    T1 is 20*D1,
    T2 is 21*D2,
    T1 < T2.

density(C,D) :-
    pop(C,P),
    area(C,A),
    D is (P*100)//A.

% populations in 100000's
pop(china,8250).
pop(india,5863).
pop(ussr,2521).
...
(19 lines omitted)
...
pop(iran, 320).
pop(ethiopia, 272).
pop(argentina, 251).

% areas in 1000's of square miles
area(china, 3380).
area(india, 1139).
area(ussr, 8708).
...
(19 lines omitted)
...
area(iran, 628).
area(ethiopia, 350).
area(argentina, 1080).
```

# VITA

Surname: Escalante Given Names: Carlos

Place of Birth: Mexico City (Mexico) Date of Birth: June 7, 1963

Educational Institutions Attended:

| | |
|---|---|
| University of Victoria | 1990 to 1992 |
| Universidad Iberoamericana, Mexico | 1981 to 1988 |

Degrees Awarded:

| | |
|---|---|
| Litentiateship [B.Sc.] (Honours) | Universidad Iberoamericana |

Honours and Awards:

| | |
|---|---|
| University of Victoria Fellowship | 1990-1992 |

Publications:

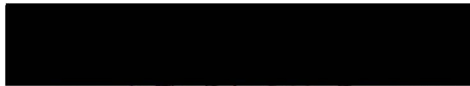
*Traductor de Notación-P a Lenguaje Ensamblador de Z80.* Universidad Iberoamericana, Mexico City, 1988.

## PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: ECTOPIC — An Extended Translator of Prolog into C++.

Author



(Signature)

Carlos Escalante
(Name in Block Letters)

September 17, 1992
(Date)