For Mary Ann and Jessica
Abstract

In this thesis we investigate a number of extensions to logic programming. In particular we look at the incorporation of concepts from functional and object oriented programming into logic programming. This is done by extending the regular syntax of logic programming to include notations for functions and classes.

The object oriented extension — class template programming — allows us to see how we might compose logic programs into larger programs. Furthermore, it gives a sound basis for understanding what a program module might be in the context of logic programming. The various features commonly found in object oriented programming languages such as message passing and inheritance are given an interpretation in this framework.

As a typical example of the power of class template programming we look at the representation of graphics in a logic programming context. This denotational graphics formalism shows how the relationship between terms and numbers can be used to construct an analogous relationship between terms and pictures. This allows us to describe pictures with terms. Furthermore, we can describe relations over pictures — such as a picture generator or the relation of 'being inside a picture' — as relations over those terms. Class templates are used as the programming formalism in which these relations can be expressed.

The semantics of our extensions are carefully examined, and in particular we prove that class templates are a conservative extension of logic programming. This gives us a sound basis for our class template language and also allows us to inherit theorems and programming techniques from logic programming itself.

We show a preprocessor which translates class template programs into efficient Prolog programs. We also investigate how a Prolog compiler might be modified to handle class templates and give even better performance than that of native Prolog.
Table of Contents

Introduction 9
  0.1 What is logic programming? 10
  0.2 Structure of the chapters 16
  0.3 Acknowledgements 17

1. Logic and equational programming 19
  1.0 Introduction and motivation 19
    1.0.1 Intended interpretations are not always neutral 19
    1.0.2 Prolog programs may not be reversible 20
    1.0.3 The proliferation of intermediate variables 22
    1.0.4 Higher order functions and relations 22
    1.0.5 Incompleteness of functional programming 24
  1.1 Conditional equalities 25
    1.1.1 Mixed relational and functional styles 27
    1.1.2 Do equations describe functions? 28
    1.1.3 Quoted expressions 29
  1.2 Higher order functions and relations 30
    1.2.1 \(\lambda\)-expressions and function values 31
    1.2.2 Set abstractions 35
    1.2.3 Loop-free programming with map 37
    1.2.4 Using the \texttt{reduce} function 38
    1.2.5 The set abstraction interpretation of logic programs 39
  1.3 Summary of equational programming 40

2. Logic and objects — the class template language 41
  2.0 Introduction and motivation 41
  2.0.1 Large programs and modules 43
    2.0.1.1 Name based modules 43
    2.0.1.2 Predicate based modules 44
    2.0.1.3 Higher order modules 45
  2.0.2 What is object oriented programming? 45
  2.1 What is an object? (from 1st principles) 46
    2.1.1 Is an object a term or a set of assertions? 47
    2.1.2 The theory of objects 48
    2.1.3 Methods in a class template 49
    2.1.4 Messages and deduction 50
## Table of contents

2.2 Class template notation 51  
2.2.1 Variable labels 53  
2.3 Class templates and modules 54  
2.3.1 More complex orderings 57  
2.3.2 Variable labels and object search 58  
2.4 Data driven programming 59  
2.5 The structure of classes — inclusion 63  
2.5.1 Broadcast messages 65  
2.6 The structure of classes — specialization 68  
2.6.1 Class rules and inheritance 68  
2.6.2 Computing with class rules 70  
2.6.3 Multiple inheritance 72  
2.6.4 Overriding inheritance 73  
2.6.5 self reference 74  
2.6.6 Error trapping and self reference 78  
2.6.7 The importance of being atomic 79  
2.6.8 super reference 82  
2.7 Mutable objects 83  
2.7.1 assert, retract and class template programs 84  
2.8 Related work 85  
2.8.1 The 'isa' interpretation 85  
2.8.2 Hayes' Logic of Frames 87  
2.8.3 The POL system 88  
2.8.4 Zaniolo's objects 92  
2.8.5 LOGIN — logic with inheritance 93  
2.8.6 Logical objects 95  
2.8.7 Objects as intensions 98  
2.8.8 Concurrent logic programming and objects 99  
2.8.9 Metalogical objects 101  

3. Denotational graphics 103  
3.0 Introduction 103  
3.0.1 Objects versus command sequences 104  
3.1 Denoting pictures by terms 105  
3.1.1 Simple pictures 105  
3.1.2 Computing with pictures 106  
3.1.2.1 A simple graphics display processor 106  
3.1.2.2 Drawing a simple picture 107  
3.2 More complex pictures 111
Table of contents

3.2.1 Aggregate pictures 112
3.2.2 Using complex pictures 113
3.2.3 Drawing complex pictures 118
3.2.3.1 Pointwise inclusion in an aggregate picture 118
3.2.4 Drawing modified pictures 120
3.3 Graphics and applications 123
3.3.1 The link between an application and its tools 126
3.3.1.1 Drawing an selecting tool icons 127
3.3.2 Displaying graphic edit windows 129
3.4 Related work 131
3.4.1 Helm and Marriot 131
3.5 Summary 132

4. Semantics of equations 133
4.1 The logic of functions 133
4.1.1 Functions, terms and canonical forms 135
4.1.2 A simple evaluator for canon 139
4.1.3 The effect of evaluation order 140
4.1.4 Compiling expressions 141
4.1.5 Quoted expressions and evaluation 143
4.2 The logic of λ's and set abstractions 144
4.2.1 Applying predicates to tuples 145
4.2.2 Giving away names 147
4.2.3 Set abstractions with free variables 148
4.2.4 Union set abstractions 149
4.2.5 Equality of relations 150

5. Semantics of class template programs 153
5.0 Introduction 153
5.0.1 The fundamental intuition 153
5.0.2 The approach to understanding 154
5.0.3 Class template programs 155
5.1 A proof theory for class template programs 156
5.1.1 The distribution of labels 157
5.1.2 Class body inference rules 158
5.1.3 Class rule inference rules 159
5.1.4 The logical connective inference rules 160
5.2 Mapping class template programs into clausal form 161
Table of contents

5.2.1 Equations and class templates 166
5.3 The soundness of class template inference 167
5.3.1 Soundness of label distribution 168
5.3.2 The soundness of the class body inference rules 169
5.3.3 The soundness of class rule inference 170
5.4 The completeness of class template inference 173
5.5 Conventional logic programs within class template programs 178
5.6 Class template programs and higher order functionals 180

6. Implementing class templates 185
6.1 A preprocessor for translating class templates into Prolog 185
6.1.1 Class bodies 186
6.1.2 Label variables 188
6.1.3 Labelled calls and class bodies 189
6.1.4 Class rules 191
6.1.5 Equations and expressions 195
6.1.6 Named functions 199
6.1.7 Object search 200
6.1.8 Representing class template programs 202
6.1.9 A comparison of class template programs with Prolog programs 204
6.2 Compiling class template programs 206
6.2.1 Choice point and call record structures 209
6.2.2 Class rules 211
6.2.3 Dynamically asserted clauses 213
6.3 Summary 213

Conclusion and future work 215

Appendix A. Syntax of class template programs 219

Appendix B. The class template preprocessor 223

Appendix C. Example class template programs 243

Glossary 249

Bibliography 255
Introduction

There are a number of reasons why logic programming is important in the field of computing. Principal amongst them is that logic programming offers the basis of a theory of computing as it might be. By this we mean that logic offers an expressive language in which we can build programs and yet which is mathematically tractable.

One of the great successes of logic programming is undoubtedly the Prolog programming language, yet paradoxically Prolog is also a great failure. Prolog strives at an ideal but does not reach it. Much of the criticism directed at Prolog has been concentrated on the so-called 'dirty' features of Prolog, condemning the cut, the use of dynamically self-modifying programs\(^1\) and so forth. However one could rightly argue that a lot of Prolog's programming power (as a practical programming language) comes from these same features.

In fact Prolog fails the ideal in many respects, not least of which is the fact that the simple flat structure of Horn clauses makes it difficult to construct large programs. In general one can argue that the more structural deficiencies such as the lack of properly constructed 'higher order' or 'meta order' aspects forced the early designers to make essentially ad-hoc decisions when these requirements arose. This has left Prolog with a collection of 'dirty' features many of which are, on reflection, simply unnecessary or can be recast into a more acceptable and still useful form.

In this work we aim to address some of the deficiencies of logic programming languages particularly in regard to the design and construction of large programs. While the ideas presented are (on the whole) in relation to logic programming, it is a simple matter to transfer them into the more practical Prolog style setting.

At the same time as enhancing the notational power of logic programming (without modifying the logic semantics that underpin logic programming) we are also able to shed light on some issues in other branches of computing; in particular the object

---

\(^1\) Because of the close connection between clauses and data in Prolog the use of assert or retract involves modifying the program itself even if the programmer's intention is simply to change a global variable.
Introduction

oriented programming paradigm. We aim to answer such questions as "What is an object?" and "What is inheritance?" in the context of logic programming. This can give us a deeper understanding of such issues without necessarily accepting unquestioningly the definitions of these concepts from conventional procedural disciplines.

In fact we choose to redefine, to some extent, these ideas in a way which is more general and powerful. So, for us an object is not simply an arbitrary collection of values and methods; instead we view an object declaratively: "an object is what we know to be true of it." This is in contrast with other approaches which liken objects to types, and the inheritance of object descriptions to type hierarchies.

Whilst we investigate extended notation for objected oriented programming (say) we wish to be careful to stay within logic. In particular we have taken care to construct a system with first order logical characterization. The result is that we have a language which is an extension of standard logic programming form, yet it is also a subset. In other words we have a language which is equivalent to normal logic programming languages but which enables us to express large scale structure more clearly.

§0.1 What is logic programming?

An interesting and difficult question relevant to us is "when do we know that we are 'doing' logic programming?" More particularly, suppose that we have a language $L$ and a mapping function from sentences in $L$ to first order clauses. Is $L$ a logic programming language? The answer in general is of course no, consider for example the case where $L$ is BASIC, and the mapping function is a BASIC compiler which compiles BASIC programs into a set of clauses. For example we might translate the BASIC program:

```
10 A = 10
20 B = 20
30 IF A<B THEN B = B DIV A
40 IF A>B THEN A = A DIV B
50 IF A=B THEN GOTO 30
60 END
```
§0.1 What is logic programming?

into the set of Prolog clauses:

\[
\begin{align*}
\text{stmt}(10,E) & : - \text{assign}(1,10,E,E1), \text{stmt}(20,E1). \\
\text{stmt}(20,E) & : - \text{assign}(2,20,E,E1), \text{stmt}(30,E1). \\
\text{stmt}(30,E) & : - \text{val}(1,E,V1), \text{val}(2,E,V2), \\
& \quad (V1 < V2 \rightarrow \text{assign}(2,V2/V1,E,E1); E1=E), \\
& \quad \text{stmt}(40,E1). \\
\text{stmt}(40,E) & : - \text{val}(1,E,V1), \text{val}(2,E,V2), \\
& \quad (V2 < V1 \rightarrow \text{assign}(1,V1/V2,E,E1); E1=E), \\
& \quad \text{stmt}(50,E1). \\
\text{stmt}(50,E) & : - \text{val}(1,E,V1), \text{val}(2,E,V2), \\
& \quad (V1 = V2 \rightarrow \text{stmt}(60,E); \text{stmt}(30,E)). \\
\text{stmt}(60,E). \\
\end{align*}
\]

augmented with suitable definitions for \text{assign} and \text{val} which respectively simulate assignment and returning values of variables in a given state. To 'execute' the BASIC program we pose the query:

\[
\text{stmt}(10,nil)?
\]

The proof of this query will follow a path which is analogous to an execution of the original BASIC program. What we have done with our BASIC compiler is to describe, in terms of logic, the semantics of BASIC programs. This in turn may allow us to analyse BASIC programs and to prove certain properties of these programs. However, we have not shown that BASIC is a logic programming language; indeed it would be surprising if we could. We will return to the reason why BASIC is not a logic programming language below.

Whilst our mythical BASIC compiler is an extreme example, there are others which are more subtle. For example the definite clause grammar (DCG) formalism is an extremely simple modification to logic programming where each grammar rule maps on to a simple clause with a few extra parameters. Yet the DCG notation is not a logic programming language either! The reason for this is that DCG's have a particular intended domain of discourse which is not sufficiently general. To see this let us look at a simple example. If DCG's were a logic programming language then we should be able to represent any clause in the DCG notation, in particular we should be able to represent the fact
likes(john, mary).

as a DCG rule. As a first approximation we might map this sentence into the rule:

likes(john, mary) --> {}.

However, the meaning of these formulae are not equivalent to each other. The grammar rule reads

"A non-terminal symbol of the form likes(john, mary) can be reduced to the empty string"

which is not at all the same as

"john likes mary"

(although [Colmereaur'87] proposes a semantic scheme for Prolog II which has very similar characteristics to this interpretation of logical sentences).

The divergence from logic programming shows up in other places within the DCG formalism. For example the sequencing connective ";" in a DCG rule is non-commutative, so the rules

nt1 --> a, b.

and

nt2 --> b, a.

are not equivalent. This shows that ";" cannot be a name for standard conjunction in DCG form. In fact there is no equivalent to conjunction in DCG's.

On the other hand it may be possible to encode sentences such as likes(john, mary) in DCG's in such a way that we simulate logic programming within DCG notation (i.e. we can construct a compiler from clauses into DCG's). However, as with our BASIC compiler, this still does not show that DCG's themselves are a logic programming language.

Another example of a class of extensions to logic programming is represented by the assortment of concurrent logic languages as exemplified by Parlog
§0.1 What is logic programming?

[Gregory’87], Concurrent Prolog [Shapiro’83] and Guarded Horn Clauses [Ueda’85]. The situation with these languages is even more difficult to determine than DCG’s. A Parlog clause does have a declarative semantics, and in this sense a Parlog program consists of sentences of logic. However the declarative reading of a program such as:

```
mode merge(?, ?, ^).
merge([E|L1], L2, [E|L3]) :-
    merge(L1, L2, L3).
merge(L1, [E|L2], [E|L3]) :-
    merge(L1, L2, L3).
```

which is ‘about’ triples of lists in a merge relation is a long way from the actual procedural reading given by the Parlog model of cooperating processes:

“A merge process can be non-deterministically reduced into a simpler merge process in one of two ways if either of the first two arguments are known.”

This is in marked contrast with Prolog (say) where the procedural reading of the merge clauses would be expressed in problem solving terms:

“A merge goal can be reduced to a simpler merge sub-problem in one of two ways (corresponding to the two clauses for merge)”

Prolog’s procedural interpretation is actually stricter (and therefore less general) than this because viewed as a theorem prover a Prolog system is not complete. There are goals which have solutions which cannot be discovered by Prolog’s simple left-right-depth-first execution strategy. However for many practical problems Prolog’s strategy is adequate for it to be viewed as a programming language as opposed to a general problem solver. Furthermore this incompleteness is regarded as a bug not a feature. Some of the solutions which are eliminated by Parlog’s proof procedure are not solutions in the intended interpretation.

The problem with languages like Parlog is that they are not honest in their

---

2 There are other logic programming language systems such as Constraint Logic Programming and mu-Prolog which have a more complete procedural interpretation.
intensions: Parlog is a language of communicating processes executing in parallel yet its syntax which is of relations fails to convey this intention. If Parlog's notation reflected this intended domain of discourse then perhaps it would be easier to decide if concurrent logic languages are logic programming languages. In this respect DCG's and BASIC are more obvious formalisms.

Whilst BASIC and DCG's are not logic programming languages, we do not wish to imply that they are not valid programming languages. In fact we feel that a programming (or any) notation should reflect as far as is possible the intended domain of discourse and DCG's in particular are a good example. DCG's are an extremely useful tool for describing grammars (whether formal grammars for programming languages or the less formal natural grammars).

One of the key features of logic programming is that it has a dual semantics: a declarative or model theoretic semantics and an operational or proof theoretic semantics. So we could take as a (rather strict) definition of a logic programming language the following:

“A logic programming language has three ‘components’: a language in which to construct formulae, a domain of discourse consisting of sets of relations which may be said to satisfy (or not as the case may be) those formulae and rules (called inference rules) for deriving new formulae from old ones. The rules must be such that they are ‘computerizable’, i.e. that sets of axioms can be viewed as programs to be executed using the rules of inference.”

The constraint that the system be computerizable means that one should be able to express algorithms in the language with the required complexity. If the system ‘converts’ an algorithm which is linear (say) into one which is quadratic (say) by virtue of its semantics it would not be fair to describe the system as a programming language.

Strictly speaking then any language which does not consist of these components alone is not a logic programming language. However we can talk of extended logic programming languages which go beyond the strict definition in some way. We identify two types of extension: conservative and radical.

Perhaps the key feature of logic is the relationship between the declarative and
§0.1 What is logic programming?

procedural semantics as identified by the notion of *logical consequence*. A new formula (as derived using the rules of inference) is said to be a logical consequence of an old formula if every model of the old formula is also a model (i.e. satisfies) the new formula. We would classify a conservative extension of logic programming to be one which maintains this relationship and a radical extension is one which does not.

Under this definition the negation-by-failure inference rule is radical unless one is careful about the declarative semantics of sets of axioms. Consider, for example, the simple program:

\[
\begin{align*}
\text{a} & . \\
\text{b} & := \neg\text{q}.
\end{align*}
\]

Amongst the models of this set of axioms are \{a, b, q\} and \{a, b, \neg q\}. However, using negation-by-failure \{a, b, q\} cannot be satisfied by the program. Much work in recent years has been in trying to find an alternative declarative semantics which incorporates negation-by-failure.

On the other hand there are other extensions of logic programming which are more conservative. We intend to examine two of these in some detail: class template programs, which are related to object oriented programming, and equations which are related to functional programming.

One might ask whether it is important that a programming language be seen to be a logic programming language. There are two different, but equally convincing, reasons why logic is important. In the first case, if we have an assertion that given languages are logic languages then we may potentially combine them. So, for example, we can combine an object oriented programming system with an equational system in the confidence that we will not get any nasty surprises.

This leads us on to the ultimate reason for having a sound foundation for our languages: we do not like to have unpleasant surprises especially not in programming languages. There are a number of features in Prolog which the careful Prolog programmer stays well clear of; for example the repeat-cut-fail combination. This (which has been likened to the Prolog equivalent of GOTO) often leads to extremely obscure programs which are difficult to debug.
Introduction

The complications arise as a direct result of the fact that the repeat-fail loop always fails, yet sometimes when it fails it represents a successfully terminated loop and at other times it has actually failed through some error or unexpected failure within the loop. The difference is often difficult to determine. Other examples abound in Prolog, and it is precisely those features of Prolog which are not well founded that can lead to problems in practice.

§ 0.2 Structure of the chapters

The text is divided into three parts. In part I we examine the language issues related to combining logic programming with functional and object oriented style programming.

Our first example of an extension to standard logic programming is the equational system. In Chapter 1 we add equations and expressions to the language in order to be able to describe naturally functions and expressions. There has, of course, been a great deal of work attempting to relate functional programming and logic programming. The content of this chapter is not fundamentally new. However our treatment of higher order functions and in particular higher order relations is not often covered. This chapter also lays the groundwork for understanding class template notation which is introduced in Chapter 2.

Our second, and main, example of an extension concerns object oriented programming. In Chapter 2 we explain the class template notation giving an informal justification of our understanding of object oriented programs. The expressive power as it relates to different programming techniques is illustrated by means of example: we cover modules, data driven programming and knowledge representation using inheritance.

Part II consists of a single chapter which is concerned with the application of class template programs. In particular we develop a major example of class template programming style as applied to the denotation of pictures. In this chapter we give a semantics to graphic terms which allows us to view graphical operations as syntactic manipulations of terms which denote pictures. This is directly analogous to the way we perform syntactic operations on number terms to denote arithmetic.
§0.2 Structure of the text

Part III is concerned with the semantics of our language and the relationships with classical logic programming. We are concerned with both the declarative semantics and the operational semantics. This last is important since we are describing a computer programming language as well as a notation for expressing ideas.

In Chapter 4 we are concerned with the semantics our the equational part of the language. We show how programs with equations and expressions are transformed into equivalent first order programs. (Equivalent in the sense that the computed answers are the same.)

In Chapter 5 we lay the foundations of the semantics of class template programs. We aim to show that clause template notation is a logic programming language.

In Chapter 6 we discuss the various aspects of implementation of class template programs. We look in some detail at a careful translation from class templates into Prolog by means of a preprocessor. The aim of this preprocessor is to generate efficient Prolog code from class template programs. The resulting code is almost as efficient as Prolog itself.

We also see how the remaining inefficiencies can be removed by modifying existing Prolog compiler technology. Such a system should actually be slightly faster for class template programs than a Prolog compiler.

§0.3 Acknowledgments

I gratefully acknowledge the support of the Science and Engineering Research Council who have sponsored my position as an Advanced Fellow at Imperial College.

The research into this area was initiated as a result of conversations with Marek Sergot who provided an early bouncing board for many of the basic ideas. Many people have made detailed comments on earlier drafts of this work including Keith Clark, Robert Kowalski, Damian Chu, Andrew Davison, Suri Sripada and Phil Vasey.
Chapter 1: Logic and equational programming

An essential part of any professional programming language is the ability to define functions and to write expressions in the text of a program. It is important because much of an application involves the description of functions and values. Nevertheless it remains the case that most Prolog compilers do not allow expressions to appear in arbitrary places within a program. In this chapter we look at one way of integrating functional programming into a logic programming system. Apart from standard first-order functions we are also able to incorporate features of higher-order programming such as lambda expressions and function variables. The result is a single formalism which combines the power of both relational and functional programming.

§1.0 Introduction and motivation

There have been many attempts to unify (sic) the functional and logic programming paradigms which seem to share many aspirations, namely declarative high-level symbolic computing. The sheer number of these alternatives (e.g. [Darlington et.al'86], [deGroot'86]) seems to be itself a worrying indication of a deep problem.

The integration of functional and logic programming is important because as programming formalisms neither functions nor relations can naturally and completely capture the intended uses of all programs.

1.0.1 Intended interpretations are not always neutral

Many Prolog programs are actually not intended to define relations. Such programs may have a non-neutral intended interpretation. For example, the famous append program in Prolog:

\begin{verbatim}
append ([], X, X).
append ([E|X], Y, [E|Z]) :- append(X, Y, Z).
\end{verbatim}
describes a relation between triples of lists, with no particular emphasis on the individual arguments. Indeed this is a favourite example to show how one program can be used for many different purposes. If asked to explain this two line program most Prolog programmers would say:

"This program appends the lists in the first two arguments to form the result in the third argument"

Of course, the append program does describe a relation and therefore has other uses, so our typical Prolog programmer would carry on:

"but append can also be used for other things like splitting lists and even searching dictionaries"

Even though a neutral interpretation of the various arguments is possible in writing the clauses for append the intended interpretation is far from neutral. The three arguments have a definite input/output relationship which is not adequately expressed by the relational syntax. We even describe the use of append for splitting lists as "using the program backwards".

Our append example is not an isolated case, in practice many, if not most, Prolog programs have an intended reading which is not adequately expressed by the neutral relational notation. This gap between the intentions of the programmer and the written formulae leads to programmers making errors and to unreadable programs.

1.0.2 Prolog programs may not be reversible

Many Prolog programs are not actually reversible in practice. This can be for many reasons, ranging from the fact that in most Prolog systems the arithmetic primitives builtin to the language are not themselves reversible\(^1\), to the more basic problem regarding the order of evaluation.

\(^1\) Though not in all. IC-PROLOG [Clark & McCabe'80] implemented a fully relational view of arithmetic where the system would guess at answers to a SUM or TIMES goal if not enough were known of the arguments to make it deterministic. Subsequent backtracking would find alternative solutions. Unfortunately the price of this generality was the restriction of numbers to the positive integers. Micro-Prolog [Clark & McCabe'84] was more modest - it implemented any input/output usage of arithmetic so long as it was deterministic.
A simple example of a non-reversible program (a program which cannot in practice be used with an inverse input/output pattern) is the program for length of a list:

\[
\text{length}([], 0).
\]
\[
\text{length}([E|L], N):-
  \text{length}(L, Nl),
  N \text{ is } Nl+1.
\]

This program really only works for two patterns of use. We can use the program to compute the length of a list, or to check that a list has a given length. But if we use it to generate a list of a given length then much useless computation is performed, and furthermore it will go into an infinite loop on backtracking when trying to find subsequent solutions.

The problem with \text{length} in reverse is not simply that Prolog arithmetic is not reversible, but that in order to generate a list of a given length the conditions in the body of the recursive clause need to be re-ordered. We also need to add a strictly redundant condition \(N>0\) in the clause:

\[
\text{length}([E|L],N):-
  N>0,
  N \text{ is } Nl+1,
  \text{length}(L,Nl).
\]

In practice few Prolog programs are reversible, and almost no programs with more than one condition in the body of a defining clause are reversible.

Although we could argue from this that it would be better have a system with a more complete execution model than Prolog (indeed there are many systems such as IC-PROLOG, mu-Prolog, CLP(R) which do have more complete procedural interpretations). However it may also be that the programmer does not intend to use his program in inverse mode. In this case the extra machinery needed to support the richer execution model is unnecessary and would be wasted.
1.0.3 The proliferation of intermediate variables.

With complex expressions there can be many intermediate variables which are not of interest except within the expression. Goal sequences such as:

\[ \ldots, \text{X1 is X+1, foo(\ldots, X1, \ldots)}, \ldots \]

are fairly frequent in large Prolog programs. Even in conventional procedural programming languages one would not have to write this, the natural way is surely:

\[ \ldots, \text{foo(\ldots, X+1, \ldots)}, \ldots \]

Expressions can be used to hide variables, this leads to less typing and fewer mistakes on the part of the programmer. Furthermore a compiler for a functional language can make assumptions about the lifetimes of these intermediate variables, and therefore optimize their implementation (using a stack for example to hold intermediate values). While in principle it may be possible for a Prolog compiler to detect such intermediate variables in a clause the recovery of this information is a difficult and expensive (in compiler time) process.

1.0.4 Higher order functions and relations

The Prolog language is traditionally a first-order language. This means that Prolog programs can be invoked but are not themselves first class objects. In higher order languages functions and relations are first class objects: a function can be the value of an expression or of a variable.

Prolog does have extensive meta-language capabilities, which allow the programmer to inspect and alter the program which is executing. These facilities are potentially more powerful (and certainly more dangerous) than a standard higher order capability. Unfortunately the utility of these meta-language features in Prolog is rather marred by their poor relationship to logic.

In some respects a higher order facility can be more expressive than the meta-logical approach: by using \( \lambda \)-expressions to denote functions we can make better
§1.0 Introduction

use of operators such as `map` and `reduce`.

For example, to append the list `[a, b, c]` to every element of the list `[[1], [2], [3]]` can be expressed as the value of the expression:

```
mp ( [[1], [2], [3]], lambda (X) * ap (X , [a, b, c]) )
```

Where `ap` is the familiar list append function and `mp` is a function which takes two arguments: a list and a single-argument function. The value of `mp` is a list whose elements are obtained by applying the mapped function to the corresponding elements of the input list.

In order to describe the same value in a Prolog program, we can use a similar primitive in Prolog called `map`. A possible definition of the Prolog `map` might look something like:

```
map([], P, []).  
map([E | O], P, [F | O]) :-  
    P (E, F),  
    map (I, P, O).  
```

In order to use `map` to solve the same query as the expression above, we would have to construct a special program `aa` (say) whose only function was to append `[a, b, c]` to its argument:

```
aa (X, Y) :- append (X, [a, b, c], Y).
```

```
{(L) | map ( [[1], [2], [3]], aa, L )}^2
```

This awkwardness is due to the fact that `map` can accept a predicate symbol\(^3\) but not the definition of a relation directly; whereas with a \(\lambda\)-expression we can describe directly the function we wish to apply to each element of the input list.

\(\lambda\)-expressions have an analogy in the relational world, called `set abstractions`, and

\(^1\) We distinguish queries that we might present to a compiler system by an expression such as: `{(t_1, ..., t_n) | C)?` where this means "print values of \(t_1, ..., t_n\) such that condition \(C\) is true". In general, a query is any formula which is of the form: `expression?`; where `expression` may be an expression to evaluate, a predication to test or a relation to compute the extension of.

\(^2\) Strictly speaking not a predicate symbol but a constant which allows one to identify a relation.
Chapter 1: Logic and equational programming

we shall see how set abstractions can be used to extend the usefulness of programs such as map.

Apart from a better use of operators such as map, allowing higher order constructs provides a possible semantic basis for program modules. This is an issue that we return to below.

1.0.5 Incompleteness of functional programming

Functional notation, on the other hand, is not complete either; there are a number of situations where functional notation is inadequate. For example, the set of equations for married below is intended to capture a database of people who are married to each other:

wife(philip)=liz.
wife(charles)=diana.
wife(tom)=mary.

This program is perfectly suited to the problem of determining the wife of a man, however an equally valid use of essentially the same information would be to determine the husband of a wife. This use amounts to using the inverse of the wife function which of course is not in general a function and is prohibited by most functional programming systems.

In this case, where the intended use of the underlying relation is genuinely neutral with respect to the arguments, the equational notation is overly constraining. In particular when representing databases a relational notation is more natural compared to functional notation.

Other examples include the don't care non-deterministic programs, such as the non-deterministic merge program we saw earlier; these programs are commonly found in concurrent logic languages such as Parlog [Gregory'87] and FCP[Shapiro'83]. Some functional systems represent this type of program by employing the concept of 'multi-valued functions': functions which can have more than one value for a given input. This is a contradiction and leads to great confusion.

In general it would be desirable to have a combined language where we use a
functional notation to describe functions and a relational notation to describe relations.

In the rest of this chapter we introduce a form of functional notation called conditional equalities. We show several examples of this notation and how expressions can be freely used in both functional and relational programs. The link between functional and relational programming is further strengthened by showing how higher order functions can be related to set abstractions which are the relational equivalent of λ-expressions.

In this chapter we concentrate on the language aspects of combining functions and relations. The intention is to illustrate the power of the various features by showing examples. In Chapter 4 we examine in some detail the semantics of functions and show how their logic can be incorporated into the logic programming formalism.

§ 1.1 Conditional Equalities

We use a system of conditional equalities to describe functions. A conditional equality is a statement of the form:

\[ f(t_1, \ldots, t_n) = G : C_1, \ldots, C_m. \quad n, m \geq 0 \]

This states that the terms \( f(t_1, \ldots, t_n) \) and \( G \) are equal whenever all of the conditions \( C_i \) hold.

The intention of an equation such as this one is to describe a function; the function being described is represented by the principal function symbol of the left hand side.

In the equation below the principal function symbol on the left hand side of the equation is double; this equation forms the definition of the double function:

\[ \text{double} (X) = X + X. \]

In a complete definition of a function there may be several equations, some of which are recursive:
Chapter 1: Logic and equational programming

fib(0) = 1.
fib(1) = 1.
fib(N) = fib(N-1) + fib(N-2).

Apart from arithmetic examples, we can also have functions over structures, lists and trees. For example, the simple function to compute the length of a list:

length([], 0).
length([E|L]), length(L) + 1.

The most famous logic program is probably append. We can also define append through equations; in this example we combine the use of operators and equations to define an infix symbol <> for appending lists:

[] <> X = X.

Some equations require the use of a conditional part to describe constraints on the application of the equation. For example, in the following equations for merge, we have to test the relative order of two elements before being able to determine which equation applies:

merge([X|L1], [Y|L2]) = [X|merge(L1, [Y|L2])]:- X < Y.
merge([X|L1], [Y|L2]) = [Y|merge([X|L1], L2)]:- X > Y.

The conditional part of an equation can also be used as a kind of where or let clause. This allows us, for example, to write the following equation for quick-sort:

sort([E|L]) = sort(L1) <> [E] <> sort(L2):- part(E, L, L1, L2).

One English reading of this equation could be

"The sort of the non-empty list [E|L] is equal to the sort of the list L1 appended to [E] and to the sort of the list L2 where L1 and L2 are obtained by partitioning L about the element E."

This combined application of conditions in equations to describe let clauses as well as conditional expressions means that our syntax can be simpler than that
§1.1 Conditional equalities

found in functional languages such as Hope [Burstall et. al.'80].

1.1.1 Mixed Relational and Functional styles

We can use functional expressions in clauses as well as within equations. This permits an integrated language with equations used to describe functions and clauses used to describe relations.

Probably the most common use of expressions in regular Prolog clauses would be to replace conditions of the form "I is I+1". For example the index program below searches a list for an element, and if the element is there it returns the position in the list at which the element was found:

\[
\begin{align*}
\text{index}(&E, [E|_], I, I) . \\
\text{index}(&E, [D|L], I, J) : - \\
&\text{index}(E, L, I+1, J) . 
\end{align*}
\]

The use of the expression \(I+1\) is so natural in this context that it almost passes without comment. In fact one of the most frequent errors novice programmers make when using Prolog is to write these types of expressions instead of using calls to the built-in predicate.

Another common situation where we use expressions is in the description of constraints, as in the case of a point being on a circle if it satisfies the equation for a circle:

\[
\begin{align*}
\text{on\_a\_circle}((X,Y), (C,D), R) : - \\
&(X-C)^2 + (Y-D)^2 = R^2 . 
\end{align*}
\]

Or two trees having the same leaf-profile if their leaf profiles are the same:

\[
\begin{align*}
\text{same\_leaves}(&L1, L2) : - \\
&\text{leaves}(L1) = \text{leaves}(L2) . 
\end{align*}
\]

\[
\begin{align*}
\text{leaves}(\text{nil}) &= [] . \\
\text{leaves}(\text{t}(L, Lb, R)) &= \text{leaves}(L) <> [Lb] <> \text{leaves}(R) . 
\end{align*}
\]
Expressions can appear anywhere in clauses, including in the head of a clause. This allows a simple definition for the increment relation, where the pair \((X, Y)\) is in the `inc` relation if \(Y=X+1\):

\[
\text{inc}(X, X+1). \quad \% \text{ expression in head}
\]

Functions may involve the use of relations in their definition. Our earlier example of the explicitly represented `wife` function may be better described through a relation `married` and a pair of equations for `wife` and `husband`:

\[
\begin{align*}
\text{wife}(W) &= W \text{ married}(W, H). \\
\text{husband}(W) &= H \text{ married}(W, H).
\end{align*}
\]

### 1.1.2 Do equations describe functions?

The syntax of equations is not itself sufficient to guarantee that they describe functions. For example in the program for `split`:

\[
\begin{align*}
\text{split}(X) &= []. \\
\text{split}([E|L]) &= [E|\text{split}(L)].
\end{align*}
\]

the left hand sides of these equations overlap; that is, more than one equation may be used to reduce an expression. Therefore, one could not say that `split` was a function. In fact it is not: possible values of the expression `split([1,2,3])` are: \([\], [1], [2] \) and \([3]\).

Not all equations with overlapping left hand sides describe relations. In these clauses for `merge` the inequality conditions constrain the equations sufficiently to make them exclusive:

\[
\begin{align*}
\text{merge}([D|X], [E|Y]) &= [D|\text{merge}(X, [E|Y])] \text{ :- } D<E. \\
\text{merge}([D|X], [E|Y]) &= [E|\text{merge}([D|X], Y)] \text{ :- } E<D.
\end{align*}
\]

If equations are genuinely overlapping then they describe a 'multi-valued function' (a contradiction in terms); in other words they describe a relation.
§1.1 Conditional equalities

Many equational programming languages impose a constraint that the left hand sides of equations should not overlap syntactically (i.e. the left hand sides of equations should not be unifiable); this tends to make programs less readable since overlapping equations must then be folded into a single equation using conditional expressions to differentiate the sub-cases. For example, if we are required to re-write `merge` so that the equations do not overlap then we have to collapse the two equations into a single one and use an `if-then-else` expression to express the alternative right hand sides:

\[
\text{merge}(\{D|X\},\{E|Y\}) =
\begin{align*}
\text{if } D &< E \text{ then }\{D|\text{merge}(X,\{E|Y\})\} \\
\text{else } &\{E|\text{merge}(\{D|X\},Y)\}.
\end{align*}
\]

Since the programmer's intention when using equations is clearly to describe a function, it is feasible for us to insist (and for the compiler to rely on the fact) that the equations do describe a function. We can legitimately regard a function as a relation with the extra uniqueness or single-solution constraint.

1.1.3 Quoted expressions

Applications which involve a degree of metalogical programming are very common in Prolog. In such circumstances we would often wish to quote terms which syntactically resemble expressions. For example, any compiler program for our equations would be required to manipulate the `names` of expressions rather than their values. In order to handle these terms we have to be able to `quote` expressions.

Our syntax for quoted terms is:

\`
E
\`

where `E` is any expression, for example

\`
(N+1)
\`

Any terms which are quoted should not be interpreted as being evaluable even if they would be so otherwise. A related operator to `\` is, of course, `unquote` or
eval. eval is a function from terms which denote expressions to the values of those expressions. Thus we have the equivalence

\[
\text{eval('E) = E}
\]

With a combination of ` and eval the programmer can selectively control the evaluation of expressions.

An interesting variation on this quoting and unquoting scheme is represented by the Common LISP [Steele'82] back-quote and comma operators. The LISP back-quote (normally written ` but we wish to avoid confusion) has the standard semantics of quote, except that a term within a quoted expression may be unquoted with the comma operator. Such unquoted expressions are evaluated in the same context as the expression which contains the back-quoted term. The effect is of providing a template or program fragment which has holes in it; the value of the whole expression is the quoted text with the holes filled in by their values.

The most significant application in LISP for back-quote is in the provision of parameterized macros. Typically the body of a macro definition is represented by a back-quoted fragment of text/structure with the substitutable sections indicated by a comma expression. A macro preprocessor is an example of a meta-program which of course has to be able to manipulate program text without evaluating it.

§1.2 Higher order functions and relations

We have seen the form and power of a first-order system of equations. In a first-order system the values of expressions are terms which can only denote data (numbers, lists etc.). We now see how to expand the types of values that we can manipulate by introducing elements of higher order functionality: \(\lambda\)-expressions and function variables.

We justify the introduction of higher order features principally on two grounds: they reflect aspects of actual programming practice and our limited uses are actually reducible to first-order anyway. We also explore an interesting parallel between \(\lambda\)-expressions (which identify functions) and set abstractions (which identify relations).
§1.2 Higher order functions and relations

An important use for these concepts is as a way of giving a sound semantics to modules and objects; this is something we shall turn to later.

1.2.1 \( \lambda \) expressions and function values

Informally, a \( \lambda \)-expression can be viewed as a function with no name. Because this function has no name it is obviously not possible to write defining equations for it. However we can use something similar to the definitional sentences we used for first-order equations.

One of the reasons that \( \lambda \) functions are so powerful is that their defining expressions need not be closed. They can contain free variables which have the effect of parameterizing the defined function:

\[
\begin{align*}
\text{e.g.} & \quad \lambda(X) \cdot X + 1 & \% \text{ fixed function - add 1 to argument} \\
\text{ } & \quad \lambda(X) \cdot X + N & \% \text{ open function - add any given number}
\end{align*}
\]

As with conditional equalities, a \( \lambda \)-expression can also have a condition. The general form of a \( \lambda \)-expression is:

\[
\lambda(t_1, \ldots, t_n) \cdot E : C
\]

where \( t_i \) are expressions, \( E \) has the same form as the right hand side of an equation and \( C \) is an arbitrary conjunction which acts as a constraint on the values of variables in much the same manner as the condition part of the conditional equation.

Variables in a \( \lambda \)-expression which are not bound by the \( \lambda \) are either bound in some enclosing \( \lambda \)-expression or ultimately in the clause/equation in which the \( \lambda \)-expression occurs; in either case such variables are considered free in the \( \lambda \)-expression. In particular a variable is considered to be free in a \( \lambda \)-expression if it occurs both within the expression and outside the expression in the clause or equation in which the \( \lambda \)-expression is embedded. Other variables which occur inside the \( \lambda \)-expression but which do not occur outside are bound or local to the \( \lambda \)-expression.
The possibility of free variables allows us to construct generic functions, and since a \( \lambda \)-expression is a term we can have functionals or function valued functions. For example the `add_n` equation below defines a functional whose values are functions which when applied to an argument add some number \( N \) to its argument.

\[
\text{add}_n(N) = \lambda(X) \cdot X + N
\]

Since we now have function values we also need to extend the standard application syntax to cope. So for example the expression

\[
\text{add}_n(3)(4)
\]

denotes the result of applying the value of the expression

\[
\text{add}_n(3)
\]

to the argument tuple

\[
(4)
\]

This is an example of the standard form of applicative notation as found in functional languages. In general the function symbol of a term or expression can be arbitrarily complex. We have not needed this complexity before in logic programming because we normally do not consider higher order notations, however it is a simple matter to justify the extension. One way is to postulate the existence of a single abstract function symbol "\( \bowtie \)" (say) in the whole language; this function symbol is not normally written explicitly but it underlies the normal surface syntax\(^5\). The standard function syntax is viewed as a concrete representation of this abstract function symbol, so a term such as

\[
f(a, b, g(c))
\]

would be viewed as the concrete representation of

\[^4\text{The value of which is, of course, 7.}\]

\[^5\text{This idea of an underlying abstract syntax with only one function symbol to represent arbitrary functions and predicates is used in LISP and in micro-Prolog [Clark & M\textsuperscript{c}Cabe'84].}\]
§1.2 Higher order functions and relations

«f a b «g c»

or more graphically as:

The more complex term

\[ f(a (g(b)) (c, d) \]

can now be simply interpreted as:

«««f a » «g b » c d»

which, when viewed graphically, looks like:

The general form of this tuple notation allows only the following form of compound term:

«t₀ t₁ ... tₙ»

where \( tᵢ \) are all abstract terms, and in concrete syntax such an abstract term is written as:

\( t₀(t₁, ..., tₙ) \)

In the case that \( n=1 \) or 2 and \( t₀ \) is a symbol for which there is an operator declaration in force then we can also use an operator notation in exactly the same
way as we can in conventional Prolog notation.

Clearly if there is only a single function symbol (albeit with arbitrarily many different arities) onto which we can map our extended term syntax then we have not actually extended the term syntax in any material sense.

Notice that we do not actually differentiate constant symbols and function symbols in this system: a conventional complex term is simply a tuple whose 0th element is a constant symbol. Furthermore the standard view of constants as being 0-ary function symbols doesn't apply either; in particular if c is a constant then \( c \cdot c() \). This seems a small price to pay for the added benefits of a higher order applicative notation.

An immediate consequence of this view of terms is the possibility of the variable function symbol (and variable predicate symbol). This is usually used to denote the application of some functional to its arguments, for example in the classic higher order function \texttt{twice} which is defined as:

\[
\texttt{twice}(F) = \lambda(X) \cdot F(F(X)).
\]

The expression \( F(X) \) (as well as \( F(F(X)) \)) has a variable function symbol \( F \). In our abstract notation \( F(X) \) would be written in the form:

\[
\langle F \, X \rangle
\]

and \( F(F(X)) \) would be written as

\[
\langle F \, \langle F \, X \rangle \rangle
\]

\texttt{twice} is a functional whose value is a function that when applied to an argument applies in turn its parameter (denoted by the free variable \( F \)) twice. If we apply \texttt{twice} to the \texttt{double} function we get:

\[
\texttt{twice}(\texttt{double})
= \lambda(X) \cdot \texttt{double}(\texttt{double}(X))
\]

The evaluation of expressions involving higher order functions can be quite difficult to follow, for example in the evaluation below:
twice(double)(3) 
= lambda(X) • double(double(X))(3) 
= double(double(3)) 
= double(6) 
= 12

We leave the reader to determine the value of the expression:

twice(twice)(add_n(2))(3) = ?

1.2.2 Set abstractions

In mathematical terms relations can be viewed as sets of tuples. In logic programming we define these sets implicitly by means of clauses. In mathematics we often use two complementary notations for sets: an explicit enumeration

\{ [1,2], [1], [ ] \}

and an implicit set abstraction which defines implicitly what a set contains rather than enumerating it:

\{ S | S<>X=[1,2] \}7

Both of these expressions denote the same set; yet the second can be viewed as a program which can be used to compute the enumeration of the set, or to determine if an arbitrary term is in the set. For our purposes we can use this set abstraction notation to represent particular sets we are interested in, namely relations; we borrow a restricted form of the mathematical set abstraction notation where the conditional part is limited to a Prolog-style conjunction. In general a logic programming set abstraction takes the form:

---

6 In the standard fixpoint semantics of logic programming the relation defined by a set of clauses can be viewed as being delimited by the fixpoints of the clauses.

7 Strictly, one should perhaps write this set abstraction as \{ S | \exists X S<>X=[1,2] \}, however in keeping with the logic programming tradition of not explicitly quantifying variables we don’t explicitly quantify the variables in this case either. However this will lead us into some complexities with regard to deciding which variables are free and which are bound by a set abstraction.
\{ (e_1, \ldots, e_n) \mid C \}\]

where \( e_i \) are arbitrary expressions and \( C \) is a conjunction of goals. Set abstractions can also be found in other functional programming languages (e.g. Hope); however, these are typically bounded set abstractions of the form:

\[ \{ E \mid X \in S, C \} \]

where \( E \) is an expression, \( S \) is a set and \( C \) is a condition. The value of such a set is the set obtained by evaluating \( E \) for each element \( X \) of \( S \) for which \( C \) holds. This form of set abstraction is strictly less powerful than the unbounded set abstraction.

The set abstraction which would be the relational equivalent of the \( \lambda \)-expression in the \textit{twice} program is:

\[ \{ (X, Y) \mid R(X, Z), R(Z, Y) \} \]

We can apply a set abstraction in the same way as a normal predicate symbol:

\[ \ldots, U = \{ (X, Y) \mid R(X, Z), R(Z, Y) \}, \ U(3, W), \ldots \]

or simply

\[ \ldots, \{ (X, Y) \mid R(X, Z), R(Z, Y) \} (3, W), \ldots \]

or even

\[ \ldots, (3, W) \in \{ (X, Y) \mid R(X, Z), R(Z, Y) \}, \ldots \]

We can have more complex set abstractions involving unions of set abstractions:

\[ \{ (X, Y) \mid \text{father}(X, Y) \} \cup \{ (X, Y) \mid \text{mother}(X, Y) \} \]

this is interpreted as the union of the two relations defined by the arms of the union, and when applied is true of a pair of arguments \((A, B)\) if either \( A \) is the father of \( B \)

\[ \text{It follows that the cardinality of the set defined by a relative set abstraction is always less than or equal to the cardinality of the governing set } S. \]
or if A is the mother of B (or both).

1.2.3 Loop-free programming with map

Compared to conventional programming languages Prolog is surprisingly short of control structures (such as while loops etc.). This is both a testament to the power of the basic rule notation and an inconvenience since loops often have to be explicitly encoded as recursions. One method for increasing the expressive power of standard Prolog (as well as most functional programming language systems) has been to use primitives such as map. A simple variety of map can be defined by the program:

\[
\text{map}(P, [], []) . \\
\text{map}(P, [E|L1], [R|L2]) :- \\
P(E, R), \\
\text{map}(P, L1, L2) .
\]

map is true of a predicate symbol P and a pair of lists L1 and L2 if the two lists are the same length and for each element E1 of L1 and corresponding element E2 of L2 the condition \( P(E1, E2) \) holds. map is sufficient to express many tail recursive loops in a higher level way, so for example to construct a loop which doubles every element of a list we can use a condition such as:

\[
..., \text{map}(\text{double}, [1, 2, 3], L), ...
\]

where L would be bound afterwards to:

\[
L = [2, 4, 6]
\]

But suppose that we wanted a loop in which 1 was added to each element of the list rather than it being doubled, then without set abstractions we would have to define a new auxiliary program, called inc perhaps, and use its predicate symbol in the appropriate map condition. However with set abstractions we can write

\[
..., \text{map}((\{ (X, R) | R=X+1 \}, [1, 2, 3], L), ...
\]
to get the value:

\[ L = [2, 3, 4] \]

A more complex example involves a union of set abstractions to express the searching of a list for a given key element:

\(..., \text{map}( \{ (X,1) \mid \text{key on } X \} | |
\quad \{ (X,0) \mid \text{not key on } X \},
\quad \{ [a,b],[c,\text{key}],[d]\}, L),...
\]

\[ L = [0,1,0] \]

1.2.4 Using the reduce function

Another loop construct in the same style of map is reduce. In this loop we reduce a list of terms to a single term, for example by adding up all the elements of the list.

As with map we can easily define reduce with a program, through equations in this case:

\[ \text{red}(F,[T]) = T. \]
\[ \text{red}(F,[T1,T2|L]) = \text{red}(F,[F(T1,T2)|L]). \]

To add up a list of numbers we can use a query such as:

\[ \text{red}(+,[1,3,5,7])? \]
\[ = 16 \]

We leave as an exercise for the reader to determine the value of the expression:

\[ \text{red}(\lambda(X,Y) \cdot [Y] <> X, [[],[a,b,c]]) \]

Higher order functions and relations provide an opportunity to be very compact and expressive with one's notation. They also provide an opportunity to be rather
opaque in one's programming style. The challenge not yet adequately addressed is how to design a notation based on operators such as \texttt{map} and \texttt{reduce} which is both programmable and powerful; perhaps this might be in the same style as the collection of operators in APL.

### 1.2.5 The set abstraction interpretation of logic programs

We can use set abstractions to give an alternative interpretation to standard logic programs. In this interpretation predicate symbols become 0-ary functions whose values are the relations denoted by the clauses and defined via set abstractions.

In conventional logic programming we use clauses used to describe a relation. In this formulation a clause becomes a statement of that tuples of a given form are members of the relation. So, for example, the Prolog program for \texttt{append} can be rephrased as:

\[
\text{append} = \{ ([], X, X) \} \mid \{ ([X|E], Y, [E|Z]) \mid (X, Y, Z) \in \text{append} \}
\]

In the same way that we can regard the clauses of \texttt{append} as implicitly defining the \texttt{append} relation, so too this equation similarly implicitly defines the same relation. Since this is a recursive equation, the meaning of the equation has to be approached via fixpoints of the equation. However it is trivial to show that fixpoints of this equation are also fixpoints of the original \texttt{append} program; in other words the declarative semantics of these equations coincide with the declarative semantics of the original logic program.

We will more use of this set abstraction interpretation of logic programming when we come to examine the semantics of class template programs introduced in the next chapter.

This set theoretic interpretation of logic programs is an also appropriate platform for the understanding of databases. In the field of databases (whether deductive or simply relational) the natural unit of data is a whole relation rather than individual tuples in the relation. Classic relational algebraic operators such as \texttt{join} or \texttt{project} compute relations in terms of other relations, and we can express these in terms of set operations and set abstractions:
join_1(R1,R2) = \{(U,V) | (X,U) \in R1, (X,V) \in R2\}

By explicitly using sets and set abstractions it may be easier to match the logic programming and database theory.

§1.3 Summary of equational programming

We have shown how to extend the regular logic programming relational syntax with a functional syntax that allows the programmer to write equations and expressions. This in particular removes one of the most common frustrations programmers have with Prolog: the interminable necessity of naming intermediate variables.

Many so-called higher-order functions can also be handled within the formalism. \(\lambda\)-expressions have a direct analogy with set abstractions, suggesting a useful extension to regular logic programming notation: we can have a higher order notation for relations as well as for functions. However, as we shall see in Chapter 4 we can transform programs with equations (including higher order functions and relations) into standard first-order programs. This means that we do not have to adjust the semantics of logic programming in any way.
Chapter 2: Logic and Objects - the class template language

In our second extension to logic programming we attempt to relate some of the key concepts from object oriented programming to logic programming. In particular we examine the class template structure of object oriented programming languages and relate it to logic programming. We shall see that there is indeed a natural relationship: one which can contribute both to the practice of logic programming and of object oriented programming.

Object oriented programming is a programming methodology which is especially suited to some programming tasks: for example discrete event simulation, implementing multiple window based programming environments and computations involving multiple worlds.

In a 'normal' programming language (such as Pascal), the programmer designs a program by separately specifying data types and a control portion of the program which operates on values. The control portion of the program is typically implemented in a procedural way: the program is structured around a control flow in which data values are seen to flow through various procedures defined in the program.

In contrast, when using an object oriented programming language, the programmer designs his programs in the context of the kinds of objects to be modelled in the system. There is often no overall control flow visible in such a program; instead there may be many smaller control flows which are embedded in descriptions of individual objects and which correspond to the local activities and operations which are allowable on the objects.

§2.0 Introduction & motivation

There are many types of application where a structure related to the objects in the physical world is appropriate. For example, we might wish to construct a database which represents the hierarchic structure of animals or of physical equipment such as an oil platform. In such a system one will naturally group data by the actions/properties of the different entities in the system rather than by the actions
and properties themselves.

In a logic program we would normally group our knowledge in terms of the *relations* involved; for example if we have a database of vehicles describing how to wash them and what fuel they consume we would typically use relations called *wash* and *fuel*:

\[
\begin{align*}
\text{wash}(\text{car, small_brush}). \\
\text{wash}(\text{lorry, large_brush}). \\
\text{fuel}(\text{car, petrol}). \\
\text{fuel}(\text{lorry, diesel}).
\end{align*}
\]

Grouping our facts in this way spreads the knowledge about individual vehicles across many relations. An alternative would be to group all the facts about each type of vehicle together:

\[
\begin{align*}
car: & \{ \\
& \text{wash(small_brush)}. \\
& \text{fuel(petrol)}. \\
& \ldots
\}
\end{align*}
\]

\[
\begin{align*}
lorry: & \{ \\
& \text{wash(large_brush)}. \\
& \text{fuel(diesel)}. \\
& \ldots
\}
\end{align*}
\]

This approach has the advantage of bringing together in one place all the relevant information about cars and lorries (say) at the cost of separating out the individual relations into many small fragments.

Computation in an object oriented program proceeds by communication, by sending messages between objects. For example, in a car-wash simulation, to ‘inform’ a \(\text{car}_X\) (say) that it is being washed the washer object will send the \(\text{car}_X\) object a ‘wash’ message:

\[
\text{car}_X : \text{wash}
\]
The complete program consists of a collection of semi-independent objects which cooperate by sending messages to and receiving messages from each other. Each object has a fixed repertoire of messages types it can handle; these are called its methods.

Other typical simulation systems for which objects form a useful paradigm are graphics and multiple window display systems. In this case the ‘objects’ model windows and pictures rather than cars and car-washes, otherwise the principles are the same: window management systems are still essentially simulation systems.

2.0.1 Large programs and modules

A key area where some of the techniques of object oriented programming languages may help is with program structure. When constructing large Prolog programs it becomes imperative to be able to group together programs with a larger granularity than that of clauses and relations. In order to allow this some Prolog designers have borrowed the concept of a module from conventional programming languages.

There are many schemes for modules in Prolog, however it is not immediately obvious what a module is. Some of the existing possibilities are name-space modules, predicate-space modules and higher-order functions.

2.0.1.1 Named based modules

A partition of the symbol name space so that the same print name in different modules is actually a different atom. Each module has a separate dictionary, names which are imported/exported are in multiple dictionaries. This is one of the earliest and simplest types of module system, and was used in micro-Prolog [Clark & McCabe'84] for example.

There are a number of problems with this style of module. In particular it is necessary to declare all constants which are to be imported and exported. This is usually somewhat tedious as the default intention is different for constants and predicate symbols: constants one wishes to be global whereas predicate symbols one usually wishes to be private.

Another problem is that it is, in practice, hard to implement a module system in
which one can redefine which symbols are imported/exported and which are local. Finally, it is not easy to allow for generic or parameterized modules where one may wish to tailor a module for different applications.

Experience with micro-Prolog’s modules suggests that a classic type of application of this module scheme is in the provision of libraries of programs or application ‘front-ends’. These are characterized by the way certain new primitives are added to the underlying system and the programmer prefers to hide implementation of the library/front-end from the user.

2.0.1.2 Predicate based modules

A similar style of module partitions only the predicate symbol name space rather than the whole symbol space. A variation of this scheme partitions the predicate and the function symbol space. A typical example of this approach to modules is in the Quintus 2.0 system [Quintus'87].

A difficulty with this style of module is that certain basic operations such as meta-call, assert and =.. have to be modified to include the module name. Furthermore, there is no perfect solution to the meta-call problem. Consider a trace of an execution in which we get the following goal sequence:

\[
\ldots, P=\text{foo}, \ldots, X=\ldots[P,a,b], \ldots, \text{call}(X), \ldots
\]

In evaluating the \text{call}(X) condition there are many places that one might look for the definition of foo: the module which is ‘current’ at the time that the \text{call}(X) condition is evaluated, the module in which the text of the \text{call}(X) condition is located, the module in which the call \(X=\ldots[P,a,b]\) is evaluated (or located), or the module where the symbol foo originally appeared in the condition \(P=\text{foo}\) (but note that in the context in which foo appears it is as a constant symbol not a predicate symbol). This last method is, in effect, the one taken in the name based module scheme, whereas typically the first method is used in this predicate based module scheme.

As with the name-based scheme, this module scheme does not immediately allow generic modules either. In fact these two schemes are better regarded as ways of splitting up large programs than methods of constructing large programs from small ones.
2.0.1.3 Higher order modules

Another way of defining modules is as higher order programs. In this view a module is a function which returns as its value selected programs from within the module. In order to call a program from another module one applies the module function to the selected predicate symbol, and then applies the result of that to the actual arguments:

..., bar(append)(A, B, C), ...

The bar 'module' could be defined through a system of equations, with the values of the function being set abstractions which define the individual relations of the module:

\[
\text{bar}(\text{append}) = \{([], X, X) | \\
\quad ([E|X], Y, [E|Z]) | \text{bar}(\text{append})(X, Y, Z) \} \\
\]

\[
\text{bar}(\text{reverse}) = \{([], []) | \\
\quad ([E|L], R) | \text{bar}(\text{reverse})(L, I), \\
\quad \text{bar}(\text{append})(I, [E], R) \}
\]

This is based on a simple elaboration of the set abstraction interpretation of logic programming introduced in Chapter 1. Although sound, and powerful, we feel that this is a rather clumsy way to write modules, requiring a substantial shift in the way people think about programs. However as the basis of understanding modules and objects this approach is interesting and we explore it in more detail in Chapter 5.

2.0.2 What is object oriented programming?

We can interpret a language such as Smalltalk [Goldberg'83] in one of two ways: as a collection of possibly interesting features, or in a more philosophical vein as a way of building programs. The distinction is important because if we take the former view then to encapsulate the features of object oriented programming languages in a logic programming context, in effect, we have to copy them in a logic programming system. We prefer a second view where we can reinterpret the various features in the context of logic programming, possibly improving them and
possibly ignoring the non-logical features.

In particular there are a number of aspects of object oriented programming languages which we do not intend to address in this chapter: these are state changes, concurrency and related issues. This is not because they are unimportant, simply that they involve issues in logic programming which are unrelated to object oriented programming, and as yet not satisfactorily resolved.

§ 2.1 What is an object? (from first principles)

There are two complementary ways of representing data in a logic program: by using terms or by using assertions. The assertional view of data is best described as the database view; for example an assertional database about people might include the following sentences about tom:

male(tom).
age(tom, 34).
married_to(tom, mary).

These sentences simply state that tom is in the male relation, that the pair (tom, 34) is in the age relation and so on. The information that we wish to describe in this situation about tom is quite heterogeneous and in a conventional database is quite likely to be spread across several relations. If the information is more structured and homogeneous we might use terms. For example lists of names and dictionary tree structures:

[mary, bill, joan]
tree(tree(nil, bill, nil), joan, tree(nil, mary, nil))

Of course, in some situations we might have assertions with lists as arguments:

children(tom, mary, [mary, bill, joan]).

Notice that the 'other' way of combining terms and assertions by having lists (or trees) of assertions is actually the basis of metalogical programming; for example we might think of having a list of assertions:

[father(john, sue),
mother(mary, bob), ...]
2.1.1 Is an object a term or a set of assertions?

The kind of information held about an individual object tends to be heterogeneous rather than homogeneous. For example, suppose that we wanted to model the famous railway trains of the world, such as the Flying Scotsman and the Transpacific express. We know that the Flying Scotsman travels at 120mph., it is green, and was made in Britain. We may also have another train in our database: the Transpacific express; the type of information that we have about the Transpacific express is similar to what we know about the Flying Scotsman but is different in detail. It is natural to represent this information through assertions rather than by terms:

speed(scotsman, 120).

speed(transpacific, 75).

colour(scotsman, green).
colour(transpacific, silver).

country(scotsman, britain).
country(transpacific, 'USA').

Considered as a notation for expressing ideas about trains, this representation may be a little clumsy since the axioms which relate to the different aspects of train-ness are spread over a large number of relations; if those relations are large (e.g. if we wanted to know the speed of many 1000's of trains) then classical notation begins to become tedious. What we wish to do is to collect together all that we know about each train and treat it as a whole.

We can do this by grouping all the axioms for each train together and giving it a label:

scotsman:

    speed(120).
colour(green).
country(britain).

}
transpacific: {
    speed(75).
    colour(silver).
    country('USA').
}

There is no requirement that we must have exactly the same information about each train; it may be that we know more about the Flying Scotsman than the Transpacific express.

This notation may be viewed as an alternative notation for standard logic programming where we 'bring out' one of the arguments into a separate label. This approach was identified by Hayes [Hayes'79] as a way of giving a logical interpretation of Frames [Charniak & McDermott'85]. Indeed Frames have some similarities to the objects in object oriented programming languages. However, as we shall see, our notation extends somewhat Hayes' formulation.

2.1.2 The theory of objects

From our point of view, a good way to look at an object is as a theory, i.e. a collection of axioms which describe what we know to be true about the object. Strictly we are interested in the set of atomic consequences\(^1\) which follow from a set of axioms. This is a similar relationship as that between a relation and the set of clauses which describe that relation. In the case of a theory it is not required that the set of atomic consequences all be about the same relation, they can be about any number of relations. We often talk about a set of axioms as if they were a theory, when what we actually mean is the set of (atomic) consequences which are derivable from the axioms.

When describing multiple theories we need to be able to differentiate them and to identify individual ones. To do this we group the set of axioms of a theory together in { }'s, and label them with a term:

label: {
    axiom\(_1\).
    axiom\(_2\).
    ...
}

\(^1\) As opposed to all the consequences which flow from a set of axioms; this would include the original axioms themselves as consequences.
The label is not part of the theory, it merely serves to identify the theory. This is analogous to the link between a predicate symbol and the relation it identifies: the predicate symbol is not part of the relation, it merely serves to identify and distinguish a relation.

We call our programs which describe atomic theories class template programs. In general a class template program has two components: a class body and a (possibly empty) set of class rules. Class bodies contain the set of axioms which are identified to be about a particular model object, and below we shall see how class rules are used to relate different class templates to each other through inheritance.

2.1.3 Methods in a class template

It is not strictly necessary to have only simple facts inside objects, we can have rules with conditions as well:

```
scotsman:{...
journey_time(Distance,T):=
speed(S),
    T=Distance+S.
}
```

Or if combined with a system of equations such as that described in Chapter 1 we could have equations inside our class bodies:

```
scotsman:{...
speed=120.
journey_time(Distance)=
    Distance+speed.
}
```

---

2 Not to be confused with normal rules occurring inside class bodies.
2.1.4 Messages and deduction

In object oriented programming languages computation is achieved by passing messages between objects. A message sent to an object is interpreted as a request to perform one of its methods.

In logic programming computation is achieved by deduction: typically by reducing a goal clause to the empty clause. The ‘allowed’ deductions are relative to the initial set of axioms; normally in Prolog this means all the axioms in the program; since we are dealing with many sets of axioms we are required to specialize each deduction to a particular labelled set of axioms.

A query to a class template program, such as to find out a particular journey-time, is written by prefixing the query condition by the appropriate label:

\{(X) | \text{scotsman:journey\_time(1000,X)}\}?  

This query is to be interpreted as:

"Display all the values of X for which the condition

\text{journey\_time(1000,X)}

is true with respect to the scotsman theory."

The label scotsman serves to identify which set of axioms to use in solving the query/condition. When demonstrating this fact only those predicates which are true in the scotsman theory will be used. If we wanted to find out how long the transpacific would take over the same distance the query would be:

\{(X) | \text{transpacific:journey\_time(1000,X)}\}?  

These queries can also be interpreted as sending messages to the scotsman and transpacific objects. By sending a journey\_time message to the scotsman object we are requesting that it perform the relevant journey\_time deduction. The rules and equations which are embedded in a class correspond to methods in conventional object oriented programming languages. We might like to call this the "message passing interpretation of logic".
Chapter 2: Logic and Objects - the class template language

If we wish to invoke a function which is defined in another class template then we can use an analogous extension to the expression syntax. If we have an expression of the form

\[ \text{label:Expression} \]

then the \text{Expression} will be evaluated using the equations defined in the program identified by \text{label}.

§2.2 Class template notation

The label that we use to identify an object theory does not need to be a constant. We can use complex terms, in particular terms with variables. This leads to great expressive power, for example, we can express a whole class of trains:

\[
\text{train}(S^3, Co, Cl) : \{ \\
\quad \text{colour}(Cl) . \\
\quad \text{speed}(S) . \\
\quad \text{country}(Co) . \\
\quad \text{journey\_time}(\text{Distance, } T) :- \\
\quad \quad \text{speed}(Sp), \\
\quad \quad T \text{ is Distance+Sp}. \\
\}
\]

The variables in the label term \text{train}(S, Co, Cl) are \text{universally} quantified over all the axioms in the class. The explicitly quantified version of \text{train} is:

\[
\forall Sp, \forall Co, \forall Cl \text{ train}(S, Co, Cl) : ( \\
\quad \text{colour}(Cl) \land \\
\quad \text{speed}(Sp) \land \\
\quad \text{country}(Co) \land \\
\quad \forall Distance, \forall T : (\text{journey\_time}(\text{Distance, } T) \leftarrow \\
\quad \quad T \text{ is Distance+Sp})
\]

In a conventional logic program if we have a clause with variables in it then the clause stands for infinitely many ground \text{instantiations} of that clause. The instantiations are arrived at by substituting terms of the Herbrand Universe for

---

3 For the sake of clarity we embolden label variables.
variables in the clause. The fact that there are infinitely many such ground versions of the clause is what differentiates first order predicate logic from propositional logic where there may only be finitely many ground formulae.

In a similar manner we can view the set of axioms which describes a particular train or a particular person being an instance of a more general set of axioms relating to trains and people in general. This more general set would of course be parameterized to allow many individual trains and people to be described. The general set of axioms corresponds to a class description, whereas an instance of the general clauses corresponds to an individual model object.

To obtain a specific set of axioms which correspond to the Scotsman from the general train theory we substitute for variables in the label (leaving local clause variables uninstantiated): Speed (which is the speed of the train) is instantiated to 120, Colour is instantiated to green and Country is instantiated to britain. With these substitutions we get the set of axioms:

```
train(120, green, britain): {
    colour(green).
    country(britain).
    speed(120).
    journey_time(Distance, Time) :-
        Time = Distance + 120.
}
```

These are the same axioms that we wrote for the Scotsman theory, therefore what is true in one is true in the other. From this point of view the two theories are identical. Of course, the label `train(120, green, britain)` is not identical to the label `scotsman` which we used above, but this is, in principle, no different to having the same relation identified by several different predicate symbols. The set of possible derivable consequences, and therefore the associated theory, is the same.

Notice that we have written the `journey_time` program slightly differently, using an extra occurrence of the $S$ global variable instead of the condition `speed(Sp)`. Variables in labels perform some of the same functions as instance variables in conventional object oriented programming languages (although they do not change state).
2.2.1 Variable labels

A powerful feature of our notation is the variable label, which is analogous to the variable predicate and variable call\(^4\) in regular Prolog.

The label in a condition/query can be represented by a variable in an axiom. At run-time the variable is bound to a term that corresponds to a label. The meaning of a variable label is compatible with the intuition that the theory identified by the variable label is the same theory as would be identified had the variable been replaced by its value in the original text of the program. For example, to determine if a given train was compatible with a given country, one might write:

\[
\text{compatible(Train,Country)} :\neg \\
\text{Country: terrain(T),}
\text{Train: suitable(T).}
\]

With each country theory containing some data about its terrain:

\[
\text{egypt: { ...}
\text{ terrain(flat).}
\text{ ...}}
\]

and each train identifying a range of suitable terrains:

\[
\text{scotsman: { ...}
\text{ suitable(T) : -}
\text{ not too_hilly(T).}
\text{ ...}}
\]

In order to determine if the scotsman train would be suitable for egypt we would pose the query:

\[
\text{compatible(scotsman,egypt)?}
\]

In solving this query we will get the following sub-goals:

\[
..., \text{Country=egypt: terrain(T), Train=scotsman: suitable(T), ...}
\]

\(^4\) Commonly mis-named as the meta-call, the variable call is a more accurate term for a condition represented by a variable in the body of a clause.
This rule for compatible trains is an example of a rule establishing a relationship between objects. Not all aspects of objects are functional; as we are interested in being able to express relationships between objects as well.

The problem of determining the exact location for such a rule raises an interesting programming methodology issue. We could put the rule with the train class template or with the country class template or even both, alternatively we could put the rule in a different class altogether. From the point of the class template programming language it makes no difference where the rule is located. We simply note that we do not force any particular solution: the programmer may put it where it is most appropriate and useful.

We will find other major uses of the variable label when we look at generic modules and data driven programming.

§2.3 Class templates and modules

We can use class templates as a notation for putting together large programs from small ones. Due to the flat structure of Prolog, large programs (e.g. those written by more than one person) can be difficult to construct simply due to their sheer size. A large program may include several thousand clauses and define hundreds of different predicates. (The programming environment in MacPROLOG is implemented as a large Prolog program with over 1200 different predicate symbols in it.) We can use the class template notation to help to construct large programs from smaller more manageable units each of which is composed of logically related definitions.

For example, a Prolog compiler may include a group of clauses which define a sort program. A cursory glance at the text of a Prolog compiler program may not immediately reveal that the auxiliary programs which are used to implement the sort program are not useful to other parts of the system. By wrapping up the sort program into a class template we can more readily identify the natural structure of both the sort program itself and the complete compiler program:
The class template for `simple` immediately identifies the auxiliary relations needed to implement the `sort` program itself. Furthermore the template gives additional information to a compiler: it can optimize the programs defined in the class template together rather than trying to optimize each relation in the program independently.

Viewed as a mechanism for grouping related definitions together class programs are already useful. Moreover by using variables in labels we can construct *generic* modules:

---

5 This is not intended as an example of an efficient sort program, rather it is supposed to be an obviously correct one.
simple(Order): {
  sort([],[]).
  sort([El|List],Sorted):-
    sort(List,Int),
    insert(E,Int,Sorted).
  insert(El,[],[El]).
  insert(El,[E|L],[E1,E|L]):-
    Order:less(E,El).
  insert(El,[E|L1],[E|L2]):-
    not Order:less(E,El),
    insert(E,L1,L2).
}

This is a generic program about the insertion sort theory. It is parameterized by a variable theory which determines (amongst other things) the ordering to be used on the elements of the list. We only need to mention the Order parameter where it is actually being referenced, where it is needed to determine if one element is less than another.

One suitable element ordering theory could be the natural numbers. We define a domain of natural numbers in terms of a set of axioms which characterizes the domain:

natural:{...}
  less(I1,I2):-
    I1<I2.
  ...}

with this definition, the query to sort a list of natural numbers is:

{(L)|simple(natural):sort([3,1,0],L)}?

which should give us the answer:

L=[0,1,3].

If we had the related class for descending, it might be:
Chapter 2: Logic and Objects - the class template language

```
descend: {
    less (I1, I2) :-
        I1 > I2.
}
```

and we could use the same sort program to sort a list in descending order:

```
{ (L) | simple(descend) : sort([[3, 1, 0], L]) }
```

which leads to:

```
L = [3, 1, 0]
```

### 2.3.1 More complex orderings

We are not limited to simple orderings such as natural and descending; we can construct more complex orderings such as one based on a cartesian product of two domains. To compare two points in the cartesian product of ascending and descending numbers (say) we would pose the query:

```
cart(natural, descend) : less((2, 5), (2, 4))?
```

Where the class template program for cartesian products contains:

```
cart(O1, O2) : {
    less((X, Y), (U, V)) :-
        O1: less(X, U).
    less((X, Y), (X, V)) :-
        O2: less(Y, V).
...
}
```

Combining the simple sort program with cartesian we can sort a list of pairs:

```
{ (L) | simple(cart(natural, descend)) : 
    sort([[3, 4], (1, 6), (3, 2), (10, 5)], L) }
```

```
L = [(1, 6), (3, 4), (3, 2), (10, 5)]
```

This ability to express different forms of ordering, effectively by composing
ordering relations in a cartesian product, is absent in most module systems of most programming languages.

It is actually only a matter of convention the way the simple class template is written so that we pass class labels as parameters rather than predicate symbols. By passing a class label as a parameter to the simple call we allow the implementer of the class template to decide which predicates/functions are needed: in the simple program the implementer needs the less relation from the natural numbers domain, whereas a slightly different implementation may require other relations. Furthermore, a more complex sort module may depend more than one relation from the ordering theory, and rather than parameterize the module with each service relation separately in the call only the imported class needs to be mentioned.

Note that in defining the simple program we only mention the 'source' of the ordering relation precisely in those parts of the program where it is needed. If we wished to parameterize a sort program written in conventional clauses we would have to add an extra parameter not only to insert but also to sort itself, and this parameter would have to be passed around the sort program so that it could be used during the insert phase. This is a notational inconvenience which makes parameterized programs distinctly more difficult to read than non-parameterized programs. Using the class template we have avoided complicating the program significantly.

2.3.2 Variable labels and object search

As we can see from above a great portion of the expressive power of class template notation comes from the ease of using variable labels. This should not be too surprising since it allows us to construct generic programs in the same manner as the variable call allows generic programs in Prolog itself.

Looking forward to Chapter 5 we can see how we can handle and justify variable labels. In that chapter we will see that in order to justify their semantics we translate class template programs into standard clauses for the class predicate. If a labelled predication in a class body has a variable label, as in:

...Order:less(X,Y),...
then, as it is translated into the class clause, we obtain the predication:

..., class(Order, Self, less(X, Y)), ...

The presence of a variable as the first element of this triple is no more remarkable than a variable anywhere else. By interpreting the labelled condition in this way variable labels reduce to terms like any other, hence eliminating the distinction.

In a practical implementation of class template programs the designer may wish to construct an indexing scheme for our abstract triples which is optimal for the normal case where the label symbol is constant or known at the time of a call. This may require that the variable label be restricted so that it must be instantiated at the time of the call (if it were not the system would generate an error).

However it is, in principle, possible to interpret variable labels where at the time of the call the label was still uninstantiated. It is instructive to see what might happen in these circumstances; assuming that the predication of the labelled condition is known then each of the class templates will be queried in turn (through backtracking in a Prolog-style system) to see if it can satisfy the condition. As a result of a successful search the variable of the variable label will be instantiated to the appropriate label. This represents a form of 'object search' in which the condition such as

..., X:colour(red), ...

where X is unbound at the time of the call might better be expressed as

“find out which objects are coloured red”

Some other combinations of logic programming and object oriented programming allow (even encourage?) this form of object search (in particular [Gallaire86]); however in our system we wouldn't recommend it lightly since the computations involved may be very expensive.

§ 2.4 Data driven programming

Another class of generic programs are the various kinds of data driven programs. Data driven programs are programs where the data being processed directly controls the kind of evaluation that takes place; in fact the data can often said to be
directly executed. Some common examples of data driven programs are arithmetic expression evaluation and meta-interpreters for logic programming.

A simple example of a generic program is an expression evaluator. Suppose that one has terms of the form:

   \( i(3) + i(5) \times (i(2) - i(4)) \)

and one is required to construct an evaluator \( \text{was} \) for these expressions. How might one do this?

In regular Prolog we can easily construct a program to evaluate these expressions, with a separate case for each type of expression:

\[
\begin{align*}
\text{was}(X+Y,Z) & : - \\
& \text{was}(X,X1), \\
& \text{was}(Y,Y1), \\
& Z \text{ is } X1+Y1.
\end{align*}
\]

\[
\begin{align*}
\text{was}(X*Y,Z) & : - \\
& \text{was}(X,X1), \\
& \text{was}(Y,Y1), \\
& Z \text{ is } X1\times Y1.
\end{align*}
\]

\[
\begin{align*}
\text{was}(i(X),X) & : - \;
\end{align*}
\]

We get into difficulties when we want to extend this program for \( \text{was} \) so that anyone can supply their own functions and have them evaluated as part of expressions in calls to \( \text{was} \).

In our extended Prolog with function and predicate variables we could write the program:

\[
\begin{align*}
\text{was}(F(X,Y),Z) & : - \;
% \text{binary operators} \ldots \\
& \text{was}(X,X1), \\
& \text{was}(Y,Y1), \\
& F(X1,Y1,Z).
\end{align*}
\]

\[
\begin{align*}
\text{was}(F(X),Z) & : - \;
% \text{unary operators} \ldots \\
& \text{was}(X,X1), \\
& \text{was}(X,X1),
\end{align*}
\]

\textbf{Note} to be confused with functional expressions.
This assumes that we also have definitions for each type of expression along the lines of:

\[ + (X, Y, Z) : \ldots \]
\[ * (X, Y, Z) : \ldots \]
\[ i (X, X) . \]

Then a new function can be incorporated without having to modify \texttt{was} by providing a definition for the new function:

\texttt{user}(X, Y, Z) : \ldots

The new definition is completely separate from the \texttt{was} evaluator, and therefore there is no need for a user to be able to access the source of \texttt{was}. However, suppose we want to be able to do more things with our expressions, differentiate them say. Along the lines of the first definition of \texttt{was} we could define \texttt{diff}:

\begin{verbatim}
diff(X+Y,DX,X1+Y1):-
    .. diff(X,DX,X1),
    .. diff(Y,DX,Y1).
diff(i(X),DX,0).
diff(v(DX),DX,1).
diff(v(DX),DY,0) :- DX#DY.
\end{verbatim}

If we also want to generalize our \texttt{diff} program to allow the differentiation of user functions then our technique of using function and predicate variables to implement the individual evaluation/differentiation functions no longer applies. This is because we have already defined the individual programs for +, * etc. to evaluate their arguments. We could change this so that we could differentiate expressions but this would be at the cost of not being able to evaluate expressions. To be able to both evaluate and differentiate expressions is difficult.

With class template programs all is not lost however. We can construct a class template for each type of expression that we may encounter. In each template (for +, * etc.) we provide methods for evaluating the expressions and we also provide methods for differentiating them:
\[ +(X,Y) : \{
\]
\[ \text{was}(Z) :-
\]
\[ X:\text{was}(X_1),
\]
\[ Y:\text{was}(Y_1),
\]
\[ Z \text{ is } X_1+Y_1. \]
\[ \text{diff}(DX,X_1+Y_1) :-
\]
\[ X:\text{diff}(DX,X_1),
\]
\[ Y:\text{diff}(DX,Y_1). \]
\]
\[ i(X) : \{
\]
\[ \text{was}(X). \]
\[ \text{diff}(DX,0). \]
\]
\[ v(X) : \{
\]
\[ \% v(X) \text{ represents a variable in the expression}
\]
\[ \text{diff}(X,1). \]
\[ \text{diff}(Y,0) :- X \neq Y. \]
\]
\[ \text{A user type of expression would also be represented by a class template, and it would contain methods for } \text{was} \text{ and } \text{diff}: \]
\[ \text{user}(X,Y) : \{
\]
\[ \text{was}(Z) :- \ldots \]
\[ \text{diff}(DX,Z) :- \ldots \]
\]
\[ \text{To evaluate an expression we put the expression to be evaluated as the label of a } \text{was} \text{ query:}
\]
\[ \{(X) | i(3)+i(5)*(i(2)-i(4)) : \text{was}(X) \}? \]
\[ \text{To differentiate it we use a similar query:}
\]
\[ \{(X) | v(x)*(i(2)+v(y)) : \text{diff}(x,X) \}? \]
\[ \text{Data driven programming is actually quite a powerful if somewhat underused} \]
programming technique for logic programming. In Chapter 3 we look in some
detail at another major example of this style of programming - graphics. As we
shall see in this case the ‘data’ are the various graphic objects and the
‘programming’ are the various calculations and operations we wish to perform on
our pictures.

§2.5 The structure of Classes - Inclusion

Objects are not always simple objects consisting of just instance variables and
methods; they often display more structure. An object can be structured in two
ways; by inclusion: it can ‘contain’ sub-objects, for example a train may have an
engine and some carriages each of which are objects themselves; and an object may
be viewed as a specialization of another kind of object. For example, a steam
train is a train whose engine uses steam. Such a train is a train like any other, but
we know more about it: a steam train is a special case of a train. We could also say
that the flying scotsman was a special case of a train.

Similarly, to describe logic theories we can use analogous ways of relating them to
each other: a theory might be constructed by including references to other theories,
and/or a theory may be a specialization of one or more other theories. We have
already seen some examples of inclusion in generic modules and in data driven
programming techniques. These concepts are important both for regular
programming but especially for knowledge representation.

Inclusion is used to describe the part of or consists of structure of an object. We
use arguments of the label which are themselves labels to denote the fact that the
incorporated labels are somehow included in the object. For example we might say
that a train consists of an engine and a set of carriages\(^7\). The label which
identifies such a train might look like:

Although a term of the form \( \{ t_1, ..., t_n \} \) is legal syntax and is very similar to the notion we
have used for a set abstraction, it actually denotes a structure of a particular form; namely a
composite term whose prefix form is:

\[
\{ \}' \{ ' , ( t_1, ..., , , t_n-1, t_n ) ... \}
\]

The set of terms \( t_1, ..., t_n \) are encoded in a single composite argument to “\{\}” involving liberal
use of the “,’” function symbol. We can do this in Prolog because the comma is an operator and
can be a function symbol as well as a lexical item used for separating terms.
§2.5 The structure of classes - inclusion

```
train(steam_engine,
    {first_cl,first_cl,
      buffet_car,
      second_cl,second_cl,
      guards_van})
```

With such a label we should be able to compute the maximum speed of a given train from the power of its engine and the load of its carriages:

```
train(E,C): {...
    speed(S): -
        E:power(P),
        C:load(W),
        S is P+W.
}
```

Each type of carriage which can be part of a train will have a description as a class template, and will contain definitions for relations such as load which characterizes the individual type of carriage. For example, we might say that a guard's van is moderately heavy and has a guard on it:

```
guards_van:{...
    load(20),
    guard_present.
}
```

A First Class carriage is heavier than the guard's van:

```
first_cl:{...
    load(35).
    ...
}
```

Whereas a steam engine might have 1000 horsepower:

```
steam_engine:{...
    power(1000).
}
```

The query to find out this train's maximum speed would be:
In order to evaluate this query the list of carriages will be 'asked' to compute their total weight by adding up the individual weights of each carriage. To do this properly we need to be able to broadcast a message to more than one object at a time.

### 2.5.1 Broadcast messages

In our list of carriages example we wanted to be able to add up all the weights of the individual carriages. There are several other things one might wish to do to our carriages. We might, for example, wish to make sure that they are all in good repair:

```plaintext
..., C \&: good_repair, ...
```

in such circumstances the whole train is in good repair only if all its carriages are, i.e. all the objects in the list agree on the message/condition good_repair.

Another type of request could be to search the carriages to make sure that there is a guard on the train:

```plaintext
..., C \|: guard_present, ...
```

In such a circumstance the condition succeeds if there is a guard on at least one of the carriages.

In general we identify three types of broadcast message to a list of objects:

1. **and-casting** where the broadcast message has to succeed for each element of the list. An and-cast condition (written with \& :)

   ```plaintext
   ..., \{E_1, E_2, ..., E_n\} \& : Q, ...
   ```
§2.5 The structure of classes - inclusion

to a list of objects is equivalent to a conjunction of queries:

\[ \ldots, E_1 : Q, E_2 : Q, \ldots, E_n : Q, \ldots \]

We can implement and-casting in a perfectly straightforward way:

\[
\{ \} \land : M. \quad \% \text{empty set always true}
\]

\[
\{ E, L \} \land : M := \begin{cases} 
E : M, \quad \% \text{send the query to first} \\
(L) \land : M. \quad \% \text{do the rest}
\end{cases}
\]

In a picture notation, one might send a list of pictures a query to make sure that they are all valid:

\[ \ldots, \{ \text{box}((20,30),(40,50)), \text{circle}((3,4),10) \} \land : \text{valid}, \ldots \]

For each primitive type of picture we have to include a definition for the valid method, for example a box is valid if it encloses a non-empty space:

\[ \text{box}(TL, BR) : \ldots \]

\[ \quad \text{valid} := \text{cart(natural,natural)} : \text{less}(TL,BR). \]

\[ \ldots \]

ii) or-casting (written using \(|:|\)) where the message has to succeed for at least one element of the list. An or-cast of a message

\[ \ldots, \{ E_1, E_2, \ldots, E_n \} | : Q, \ldots \]

to a list of objects is equivalent to a disjunction:

\[ \ldots, (E_1 : Q; E_2 : Q; \ldots; E_n : Q), \ldots \]

As with and-casting, we can implement or-casting through a simple program:

\[
\{ E, L \} | : M := E : M. \quad \% \text{does msg work?}
\]

\[
\{ E, L \} | : M := (L) | : M. \quad \% \text{try the rest}
\]

\[
\{ E \} | : M := E : M. \quad \% \text{last item in the 'set'}
\]

Again, in a picture system, one may interrogate a list of pictures to see if a point is
inside the list, i.e. inside one of the components of the list:

\[ \ldots, \{ \text{box}((20,30),(40,50)), \text{circle}((3,4),10) \} \]
\[ \text{\ldots, ptin}((10,20)), \ldots \]

Each type of picture would have a method in it for determining if a point is in the picture:

\[
\text{box}(\text{TL}, \text{BR}): \{ \ldots \\
\quad \text{ptin}(P) :- \\
\quad \quad \text{cart}(\text{natural}, \text{natural}): \text{less}(\text{TL}, P), \\
\quad \quad \text{cart}(\text{natural}, \text{natural}): \text{less}(P, \text{BR}). \\
\quad \ldots \}
\]

iii) The third type of broadcasting can be called \textit{map-casting}. This is in general harder to generalize than either and- or or-casting. It is similar in spirit to the \textit{map} operator we saw in Chapter 1 to map a list of arguments to a list of answers:

\[
\text{map\_cast}(\{\}, \text{M}, \{\}). \\
\text{map\_cast}(\{E, L\}, \text{M}, \{V, VL\}) :- \\
\quad \quad E: \text{M}(V), \\
\quad \quad \text{map\_cast}(L, \text{M}, VL). \\
\text{map\_cast}(\{E\}, \text{M}, \{V\}) :- \\
\quad \quad E: \text{M}(V). \\
\]

We can now see that our earlier attempt to define \textit{speed} was a little over-simplified since the condition

\[ \ldots, \text{C:load}(W), \ldots \]

is actually better described via a broadcast condition, so we combine map-casting with the use of a \textit{reduce} operator, to produce the rule:

\[
\text{train}(E, C): \{ \\
\quad \quad \text{speed}(P+W) :- \\
\quad \quad \quad E: \text{power}(P), \\
\quad \quad \quad \text{map\_cast}(C, \text{load}, WL), \\
\quad \quad \quad \text{reduce}(+, WL, W). \\
\quad \}
\]
map_cast is actually better viewed as a function, from a list of objects to a list of values:

map_cast (\{\}, M) = \{\}.
map_cast (\{E, L\}, M) = \{E: M, \text{map}\_\text{cast}\ (L, M)\}.
map_cast (\{E\}, M) = \{E: M\}.

We could have written the definition of speed as an equation:

\[ \text{speed} = E: \text{power} \ast \text{red} (+, \text{map}\_\text{cast}\ (C, \text{load})) \].

§2.6 The structure of classes - Specialization

Another important way that we noted of structuring our knowledge into classes is through specialization. For example, if we want to describe classification in the animal kingdom then we will need to be able to express knowledge of the form:

"A bird is a two legged animal with feathers that flies"

and

"A penguin is a flightless bird"

It would be very convenient if we could directly express this knowledge in a simple way. If we wanted to determine that a bird can run (as opposed to fly) then we could do so by checking whether animals run; even though we directly know that birds fly, we can indirectly deduce that they run.

We are not limited to animals for the need to specialize; other examples from conventional computing include multiple window environments (an edit window is a special case of text window which is in turn a type of window...) and the description of pictures.

2.6.1 Class rules and Inheritance

The logical equivalent of specialization is inheritance. When we say that a bird is a special case of animal we are stating that whatever is true of animals is also true of
birds: the bird theory inherits a set of consequences from the animal theory.

We express this kind of relationship between classes by means of *class rules*:

\[
bird \leq animal.\]

Informally this could be read as "animal implies bird": if something is true of animals then it is also true of birds.

Of course there may be other facts which are true of birds other than what is true of animals. Class rules can be combined with *class bodies* to express the special nature of birds:

\[
bird: \{ \begin{align*}
&\text{no_of_legs}(2). \\
&\text{mode}(\text{fly}). \\
&\text{covering}(\text{feathers}). \\
&\end{align*} \\ 
\}
\]

\[
bird \leq animal.\]

This combination of class bodies and class rules is reminiscent of the way a single relation is often defined. The class body is analogous to the collection of assertions which might define the base cases of an individual relation, whereas the class rule is analogous to a rule. The difference is one of scope: a normal axiom (rule or assertion) declares which individual *tuples* belong in a given relation, the class bodies and class rules declare which *relations* are in a given theory. Another difference is that class rules are not usually recursive, although there is no prohibition on recursive class rules.

Class rules can be used to give names to particular instances of objects, for example to say that "tom is a male person of age 33" we could use a class rule:

\[
tom \leq \text{person}(\text{male}, 33)\]

Using class rules in this way allows us to reestablish the links between special names and general sets of rules.

For example, the class rules
(re-)define the theories associated with scotsman and transpacific in terms of the general train theory. In terms of what is true about the scotsman, (or the transpacific), there is no difference between using a class rule and a general class body about trains, or having a specific class body about each type of train.

In general, a class rule of the form:

\[ l(l_1, \ldots, l_k) \leq m(m_1, \ldots, m_m) \]

where the \( l_1, \ldots, l_k, m_1, \ldots, m_m \) are all expressions, states that any ground fact in the theory identified by the label \( m(m_1, \ldots, m_m) \) is also in the \( l(l_1, \ldots, l_k) \) theory. As with a conventional clause, all the variables in the class rule are assumed to be universally quantified.

### 2.6.2 Computing with class rules.

A class rule expresses a way to replace the label in a condition; this is analogous to the regular rules in class bodies which replace conditions by conjunctions of new conditions. So if we have a condition of the form:

\[ \ldots, \text{bird:breathes}(X), \ldots \]

by using the class rule for bird being a special case of animal, in order to solve the condition we can replace the bird label by the animal label:

\[ \ldots, \text{animal:breathes}(X), \ldots \]

The proof proceeds as though the original label were actually animal; the whole condition succeeds if breathes(X) is true in animal. Usually, of course, there will be axioms in the animal class which define the meaning of breathes; however there could be another class rule for animal which postpones the call to
In general, whenever we have a condition of the form:

$$\ldots, \text{lab}(l_1, \ldots, l_k) : p(p_1, \ldots, p_n), \ldots$$

we can proceed either with a class rule which replaces the label with a new one, or we can use a rule from within a class body to reduce the predication to a conjunction keeping the label the same.

We can see this by tracing a simple query from the personal relationships program:

```prolog
bob: {
   i) likes(X):-
       X:likes(logic).
}

ii) peter<=person(male,30).

person(S,A):{
   iii) sex(S).
   iv) age(A).
   v) likes(logic):-
       age(Ag),
       Ag<40.
}

vi) person(S,A)<=universal.

A typical query could be: does bob like peter?

bob:likes(peter)?

peter:likes(logic)? because bob likes anyone who likes logic (i)

person(male,30):likes(logic)? because peter is a male person (ii)

person(male,30):age(Ag),
    person(male,30):Ag<40? young(!) people like logic (v)

person(male,30):30<40? by (iv)
universal: 30 < 40? universal has the system relations in it (vi)

Q.E.D.

2.6.3 Multiple inheritance

In our bird example we had a local definition for mode of travel which stated that birds can fly. In general animals may have many modes of travel (e.g. walking, hopping, running etc.). We can have statements in the class body for animal which include other modes:

\[
\text{animal: \{ ... mode(walk). ... \}}
\]

The question arises, given the fact that a bird is an animal and the query:

\[
\{ (X) \mid \text{bird:mode}(X) \}\?
\]

what are the valid values for X which satisfy the query?

\[
\begin{align*}
X &= \text{fly} & \text{% from class body for bird} \\
X &= \text{walk} & \text{% from animal}
\end{align*}
\]

This is a degenerate example of multiple inheritance. In general all ways of deducing an answer are valid and therefore all answers are correct. Another way to get multiple answers is by having several class rules. For example, apart from being an animal a bird may be an aeroplane:

\[
bird \leq \text{aeroplane}.
\]

In which case we are likely to get the further answer:

\[
X = \text{taxi}
\]

to our query.
Since we allow (indeed encourage) multiple inheritance we are left with the problem of deciding which method to use when there are two or more independent definitions for a method. In conventional object oriented languages this problem is 'resolved' by imposing a standard search order: the ancestors of a class are searched in some predetermined order and the first one which contains an implementation of the method is used. The search order is usually based on the order of declaration of ancestors in a class template; a less than clear situation for many programs. (In any case the programmer has the difficulty of arranging his declarations so that the right method implementation is invoked, and where more than one method is involved this may not be possible.)

The problem of multiple inheritance is much less severe in a logic programming language since it is not required that there be a single definition for a relation. Nor is it required that a method be deterministic. Thus in a logic programming language we would normally inherit all the possible definitions of a method from all the ancestors of a theory. A Prolog-style system would use backtracking to search all the possible methods until one was found that worked.

Of course while it may be convenient for an object to have multiple inheritance from different sources, we may sometimes wish to restrict inheritance to only one source— for example. Or perhaps more often we will wish to override a derived inheritance with an explicit definition in a class body.

2.6.4 Overriding inheritance

Suppose that we wanted to describe a theory of penguins, these are birds; on the other hand they cannot fly so the simple class rule:

\[ \text{penguin} \leq \text{bird} \]

is insufficient since this would allow us to conclude that penguins can fly. This would follow even if the class body for penguin included a local definition of mode which does not mention flying.

What we need to be able to express is that penguins are birds, but they cannot fly. In other words whatever is true of birds is true of penguins except anything about a penguin's mode of travel.

We can express this by using a second form of class rule called the overriding
§2.6 The structure of classes - specialization

**class rule**\(^8\). Written in the same way as a regular class rule except that we use the `<<` connective as opposed to `<=`. The meaning of a class rule such as

\[\text{penguin} << \text{bird}.\]

is

"Any relations defined in the \(\text{bird}\) theory are also part of \(\text{penguin}\) theory, *except* those relations which have a local definition inside a class body for \(\text{penguin}\), these latter relations *overriding* relations inherited from \(\text{bird}\)."

On its own using a class rule of this form would not be sufficient to eliminate the penguin's ability to fly unless we also include a local definition for \(\text{mode}\) in the class body for \(\text{penguin}\). This local definition has the effect of completely redefining the \(\text{mode}\) predicate in \(\text{penguin}\) so that we now have the opportunity to omit flying. This new definition may still be augmented in the normal way by other class rules if these are of the normal type.

To provide a complete definition of \(\text{mode}\) could become awkward if the original definition were inherited from a large number of class templates. In this case we should reestablish the inheritance links specifically by adding extra class rules. In general this kind of complexity is rare in normal programming practice although more common when trying to represent knowledge bases.

Clearly this form of class rule also introduces an element of non-monotonic reasoning: by adding true statements to the \(\text{penguin}\) theory we may remove consequences from the \(\text{penguin}\) theory which were previously true. As we shall see in Chapter 5 this non-monotonicity reduces to negation-as-failure with the added bonus that class template programs tend naturally to be stratified.

### 2.6.5 self reference

In our class body for \(\text{animal}\) above we had a definition for \(\text{mode}\) of travel that included

\[\text{mode}(\text{walk}).\]

\(^8\) This form of **Class rule** was suggested to the author by means of a private communication, Luis Monteiro, March 1988.
which is true for most animals. Other modes of travel which are common amongst animals include running and galloping. By observing the natural world one might conclude that only two-legged animals can run and only four legged animals can gallop. How can we express this fact?

We could have extra rules in each sub-class of animal of the form

```prolog
mode(run).
```
when it is a two-legged animal, and

```prolog
mode(gallop).
```
when it is four legged. However, by observation, we have determined a general rule for running and galloping which should be expressed explicitly, so how can we express the rule (preferably in the animal class)?

The natural rules for mode in the animal class body are:

```prolog
mode(run) :-
   no_of_legs(2).
mode(gallop) :-
   no_of_legs(4).
```

The no_of_legs property is not fixed for all animals, nor in this case is it a parameter of the animal label. For us to be able to refer to the number of legs animals have we could add an extra parameter to the animal label. This would require us to change all our class rules and labels which define different types of animal to add the extra parameter, furthermore the number of legs that an animal has might not be a simple constant: it may require computation (does a monkey use two, four or five legs (its tail) when swinging through trees?).

Our rule for mode of travel is true for all types of animals and therefore it is true of the animal class; on the other hand this rule depends on sub-relations which are specific to each class of animal which cannot themselves be encapsulated inside the animal class.

To resolve this conflict we use self reference. self is a keyword which at all
times identifies the original label, i.e. whenever an explicitly labelled condition appears in a rule or query (including the variable label case), then self will refer to the label in the text of the program. In particular self is invariant under the use of class rules. The effect of self is to 'root' us back into the original class template no matter what the current class is.

Using self the rules for mode of travel become:

\[
\begin{align*}
\text{mode}(\text{run}) : & - \\
& \quad \text{self:}\text{no_of_legs}(2).
\end{align*}
\]

\[
\begin{align*}
\text{mode}(\text{gallop}) : & - \\
& \quad \text{self:}\text{no_of_legs}(4).
\end{align*}
\]

So, to find out if birds run or gallop we use a condition of the form:

\[
\ldots, \text{bird:mode}(X), \ldots
\]

which can be reduced thus:

\[
\begin{align*}
\ldots, \text{bird}_9 : \text{bird:mode}(X), \ldots & \\
\ldots, \text{animal}_9 : \text{bird:mode}(X), \ldots \\
\ldots, \text{self}_9 : \text{bird:no_of_legs}(2), \ldots & \quad \% X=\text{run} \\
\ldots, \text{bird:no_of_legs}(2), \ldots \\
\ldots, \text{true}, \ldots
\end{align*}
\]

If we were to ask a similar question of horse (say) then the trace would show that self was bound to horse which has 4 legs, and which therefore gallops.

We can also use self as a normal term. Suppose that we had the following program about tom and jane:

i) \[ \text{tom}\leq \text{person}(\text{male, 33}). \]

ii) \[ \text{jane}\leq \text{person}(\text{female, 2}). \]

These class rules state that tom and jane are examples of \text{person: anything}.

\[ \text{We show the original self as a subscript on the label.} \]
which is true of 33 years old male person is also true of tom and anything which is true of 2 year old females is true of jane.

In the class body for person we wish to indicate that people like children, and that people like those who like them. The first can be stated as:

\[
\text{person}(\text{Sex, Age}) : \{ \ldots \\
\quad \text{age(Age)}.
\]

\[
\text{like}(P) : - \\
\quad P: \text{age}(A), A<10.
\]

The second involves the self keyword:

\[
\text{like}(P) : - \\
\quad P: \text{like}(\text{self})^{10}.
\]

To see if jane likes tom we get the trace:

\[
\text{jane: like(tom)?} \\
\text{person(female, 2): like(tom)? by ii) } \\
\text{tom: like(\text{self}=jane)? by v) } \\
\text{person(male, 33): like(jane)? by i) } \\
\text{jane: age(A), A<10? by iv) } \\
\text{person(female, 2): age(A), A<10? by ii) } \\
\text{2<10? by iii) } \\
\text{Q.E.D.}
\]

This use of self is not so easily replaced by adding extra parameters to labels; although self can best be understood in terms of an implicit extra parameter in every label. This parameter is passed unmodified by class rules.

\[
\text{penguin(\text{Self}) <= bird(\text{Self})} \\
\text{bird(\text{Self}) <= animal(\text{Self})}
\]

and any conditions in a class body which have an explicit label have that label repeated as a self reference within the explicit label. For example the condition:

\[
\text{penguin(\text{Self})) <= bird(\text{Self})} \\
\text{bird(\text{Self}) <= animal(\text{Self})}
\]

We ignore in this example the fact that a Prolog style system could easily loop with a symmetric definition like this one.
§2.6 The structure of classes - specialization

..., penguin:mode(X), ...

is actually the condition:

..., penguin(penguin):mode(X), ...

An occurrence of the self keyword in the class body is replaced by another occurrence of this implicit Self variable.

The condition:

..., self:no_of_legs(2), ...

becomes the condition:

..., Self:no_of_legs(2), ...

Self reference has interesting implications when it comes to understanding a class template program. If a program refers to self then it can only have a complete meaning in terms of its various specializations: in the case of animals one would never expect to get a condition of the form:

..., animal:no_of_legs(X), ...

other than by inheritance! A direct reference has no obvious meaning.

2.6.6 Error trapping and self reference

In many Prolog systems it is possible to define a special program which is invoked in the case of an error. For example, if a program is called when there is no definition for it then instead of simply failing (or stopping), the user-defined error handler may be invoked. This error handler program may then be able to repair the damaged call by loading in library definitions or editing the call. Error handlers were first introduced in MProlog [Bendl’80], and micro-Prolog [Clark & McCabe’84]; they are not in every Prolog system but they are undoubtedly useful for large-scale applications.

In a large Prolog program one may have many different situations where errors can
arise; furthermore the desired response to a given error may be different in different circumstances (for example sometimes it may be highly desirable that the system produces no visible output to avoid disrupting an ongoing display). Error recovery is especially complicated when there are many modules in the system. Unfortunately most systems only allow for a single global definition of the error handler (this is due to the complexities of trying to decide which error handler to call when there are many possible alternatives).

In class template notation it is perfectly possible to allow each class template to have its own error handler. When an error occurs the error handler associated with the class template of the offending condition is invoked. To allow for the possibility of the current label arising through inheritance we would actually employ the self concept to help decide which error handler program to use. So, if the call

\[ ..., \text{foo}(t_1, ..., t_n), ... \]

raises an error, then the system (dynamically) replaces it by the call:

\[ ..., \text{self:error(error_code, foo}(t_1, ..., t_n)), ... \]

By using the self keyword the system can ensure that whichever error handler is available (often either the one in the current class template or the default system one), it can be invoked correctly.

### 2.6.7 The importance of being atomic

We have stated earlier that inheritance is of an atomic nature (i.e. when we inherit via a class rule we import the relations defined in the class template rather than the definitions themselves). We have not really justified this interpretation. It must be said that some rather unexpected consequences flow from this decision. For example, we could have the following axiom in the class body for person:

```prolog
person(A) {...
  childish :- likes(toys),
...
}
```

Given our definition above of a child being a young person, a condition of the form:
§2.6 The structure of classes - specialization

..., child(10):childish,...

would (unexpectedly?) fail since the proof of the condition

..., likes(toys),...

which is necessary to establish childishness takes place in the context of the person class not the child class. For our rule to be correct we have to write it with a self keyword as:

childish : self:likes(toys).

With this version of the childish definition the question of whether a child likes toys is forced, via the self keyword, to be relative to the child class (and is therefore true in this case).

This 'problem' is a direct result of our interpretation of inheritance, which relies on the rather subtle definition of inheritance that only atomic consequences of a theory are inherited via class rules. If all the consequences of the theory were inherited then we would have to include the original axioms of the host theory; in particular the childish rule from person would also be a rule for the child class and we would not need to use the self keyword here (indeed it becomes almost completely redundant).

In fact this form of inheritance amounts to unioning the set of axioms belonging to the host theory with the local axioms defined in the class body. This unioned set forms the basis of the complete set of consequences of the theory.

Unfortunately other unexpected results flow from an interpretation of inheritance which is based on unioning sets of axioms. Two programs in separate class templates may start to interact in unexpected ways through being inherited. This is especially inconvenient in the case of stratified programs involving negated conditions where the stratification may break down. In the following example, we have two class template programs t1 and t2 each of which is stratified\textsuperscript{11}, but a further class template program t3 which is defined solely in terms of inheritance of t1 and t2 is only stratified if inheritance is defined as inheritance of atomic consequences:

\textsuperscript{11} I.e. no predicate depends, for its definition, on a negated recursive call to itself.
Chapter 2: Logic and Objects - the class template language

\[ t_1: \{ \ldots \]
\[ p := q, \neg r. \]
\[ r. \]
\[ \ldots \} \]

\[ t_2: \{ \ldots \]
\[ r := s, \neg p. \]
\[ p. \]
\[ \ldots \} \]

\[ t_3 \leq t_1. \]
\[ t_3 \leq t_2. \]

Under the 'all consequences' view of inheritance we effectively union the axioms of \( t_1 \) and \( t_2 \) to obtain a set of axioms for \( t_3 \), so we could just as well have defined \( t_3 \) directly:

\[ t_3': \{ \ldots \]
\[ p := q, \neg r. \]
\[ p. \]
\[ r. \]
\[ r := s, \neg p. \]
\[ \ldots \} \]

The clauses for \( p \) in this class body contain negated conditions which mention \( r \). However the definition of \( r \) also refers to a negated \( p \) condition. Thus \( p \) and \( r \) rely on each other via negated conditions and by definition are not stratified. That a program is stratified is a fundamental property of safety that many systems rely on.

The damage caused by this kind of unexpected interaction is potentially far worse than not being able to derive all non-atomic consequences of inherited theories since the former affects the safety of programs involving negation. In any case we have seen how the judicious use of the \texttt{self} keyword can have the \textit{effect} of inheriting all the necessary consequences; furthermore this is under programmer control. As we shall see below the translation of the \texttt{self} keyword is itself relatively straightforward.
2.6.8 super reference

The complement of self reference is super reference. Like self, it is a keyword which in this case can only be used as a label.

super is used when we wish to explicitly avoid using a local definition and to only use an inherited definition. Recall that by using the overriding form of class rule any local definitions of a relation can override any definitions which would otherwise be inherited through the class rule. The super keyword is especially useful when one wishes to redefine or augment a relation defined in an ancestor class.

Suppose we had a system of classes which relate to windows, in a multiple window display system. We would have classes for windows and for labelled or titled windows:

\[
titled\_win(Title,Pos,Size) <= window(Pos,Size).
\]

When a titled window is moved it is desirable to move the label at the same time that we move the window itself; to implement this one would define a local version of move for titled_win which performs the required adjustment. A first approximation to this version of move might be:

\[
titled\_win(Title,Pos,Size):
\]
\[
\{ \ldots
\begin{small}
move(A):-
\begin{align*}
&\text{window}(Pos,Size):move(A),
&\text{move\_title}(A).
\end{align*}
\end{small}
\}
\]

The problem with this is that it requires us to know that a titled_win is a special case of window. Since titled windows might also be special cases of other class templates, for example a titled window could be a graphic region or a process, it may not be clear which super-class to invoke to complete the move. super solves that for us:

\[
move(A):-
\begin{align*}
super:move(A),
\end{align*}
\]
The super reference 'says' to use only the inherited definitions of move and not to use any local definition. Whichever super-class has the definition of move will be used to move the window. Logically it is as though for every label there was a new label\(^\dagger\) which had the same class rules as the label but no class body. This new label is used whenever the super keyword is encountered.

Super references make it easy to construct specializations where a relation is defined as a modification of, as opposed to a strict addition to, a relation defined in an ancestor, as in the case of moving labelled windows.

§2.7 Mutable objects

In a conventional language such as Smalltalk objects generally have state. A Smalltalk object has local variables (the instance variables) which can change over time. Any objects which own references to the changing object must always 'see' the latest version of the object. These are so-called mutable objects as opposed to immutable objects which do not change over time. At first sight it may seem that we are not able to represent mutable objects in our class template notation.

Since labels are just terms, we can manipulate them, and we can get some of the effect of mutable objects by allowing programs to return new labels. So, for example, if we wish to change the speed of a train we can do so by:

```plaintext
train(Sp,Cl,Cn): {
    change_sp(Newsp,train(Newsp,Cl,Cn)) .
}
```

A call of the form:

```plaintext
..., train(90, green, britain): change_sp(200, New), ...
```

results in the variable New being instantiated to the label term train(200, green, britain); this can be used in a subsequent call to find out the length of time a 1000 mile journey will take at 200 mph:
§2.7 Mutable objects

..., New: journey_time(1000, Time), ...

This ability to return new labels as a result of some computations is not exactly the same as having mutable objects since we are not actually changing an existing object, merely creating a 'new' one. This means that all the relevant consumers of the train at its new speed must be given the new label explicitly, otherwise there could be an inconsistency in the interpretation of the train's speed.

Note that since two theories are equal if their class labels are equal it is not necessary to have an explicit mechanism to create new objects; a class label is simply a normal term which can be generated through the normal process of unification.

One possible route to modelling objects with state is to use a process communication model along the lines suggested by Shapiro [Shap'83]. In such a model objects are processes rather than simple labels, however this is a complicated approach and does not really solve the underlying issues.

2.7.1 assert, retract and class template programs

In an actual implementation of class template programs it may be necessary to perform assignments, in much the same spirit and manner as assert and retract are used in real Prolog systems (as opposed to pure logic programs). Of course any such programs which actually use assert and retract are hard to justify logically but we do not concern ourselves with this 'detail' in this discussion.

Normally assert and retract have as arguments terms which describe the clauses to add or remove from the dynamic part of the program. When modifying class template programs it is important to identify which class is being modified. the obvious way of doing this is to add an extra label term argument to assert and retract which identifies the class.

As a clause is asserted a label term is associated with it. Clauses which have been asserted are only accessible if the label of the condition unifies with the label associated with the asserted clause. This allows different instances of labels

\[12\] It should be noted that, from the point of view of assignment style programming, using assert and retract to implement assignment is not especially elegant; nor is it efficient. However efficiency of assignment is not one of our concerns!
to have 'local' versions of asserted clauses and these clauses will not clash in the sense of unifying when they should not.

A very common label to use when asserting clauses is likely to be the self label. Using self as a label of an asserted clause allows for the maximum differentiation between different instances of labels. One could even use the convention that if the 'old' versions of assert, retract and clause are invoked without an explicit label argument then self is assumed.

§ 2.8 Related work

The investigation of object oriented programming and its relationship to logic programming has been quite an active field in recent years. The published work falls into two distinct categories — the work of people such as Hayes and other in which the main effort is to fit some of the concepts of object oriented programming into logic without compromising logic. We would like to believe that our work also fits into this category.

The second group concentrates on attempting to implement concepts of object oriented programming in logic programming (or some variation thereof). This work is characterized by the embracing of assignment and much effort is concentrated on modelling assignment in a logic programming context. An interesting example of this is the work by Conery on Logical objects which we examine below. Others have used the committed choice logic programming languages such as Parlog and Concurrent Prolog to model assignment. Below we look at a number of published accounts of work related to logic and object oriented programming.

2.8.1 The 'isa' interpretation

One of the classic ways in which logic programmers have described objects (and in particular inheritance) is through isa axioms [Kowalski'79]. In this formulation we would express the fact that birds can fly (for example) with the rule:

i) can_fly(X) :- X isa bird.

Similarly, we would could also express that animals can walk:
ii) \[ \text{can\_walk}(X) :- X \text{ isa animal}. \]

Inheritance is expressed by means of rules about the \textit{isa} predicate such as

iii) \[ X \text{ isa animal} :- X \text{ isa bird}. \]

which expresses the assertion that all birds are animals. If we want to find out if birds can walk we postulate the existence of a bird \textit{tweety} (say):

iv) \[ \text{tweety} \text{ isa bird}. \]

and pose the query

\[ \text{can\_walk(tweety)}? \]

If we follow a proof of this query we can see how the rules for \text{can\_walk} and \text{isa} are used to solve the goal:

\[ \begin{align*}
& \text{can\_walk(tweety)}? \\
& \text{tweety} \text{ isa animal?} \quad \% \text{ by ii)} \\
& \text{tweety} \text{ isa bird?} \quad \% \text{ by iii)} \\
& \text{Q.E.D.} \quad \% \text{ by iv)}
\end{align*} \]

This approach is quite different to ours, and (in our opinion) suffers from two defects. The first stems from the fact that in the rules for \text{can\_fly} and \text{can\_walk} etc. we explicitly mention a term (bird, animal) which denotes the set of all birds (animals etc.). Although we managed to avoid actually writing down the set of all birds (animals) by using a constant to denote it we are nevertheless using a set theoretic formulation of objects: we are forced to explicitly name the sets of objects that we are interested in. One consequence of this approach is that when we wish to ask a general question such as "Can birds walk?" we have to postulate the existence of at least one bird (tweety) before we can even ask the question. In fact this aspect of the \textit{isa} formalism is shared with some conventional object oriented systems: when a new object is created it must be given a name and no operation can be performed without at least one object being created.

On the other hand in class template programs we do not need to identify explicitly the set of all birds (or animals). A class label is simply a means of identifying certain programs and relations; in particular a label does not denote the relations.
Chapter 2: Logic and Objects - the class template language

We do not have to have a term which denotes the set of all birds (or animals) but instead our label identifies what is true of birds (and animals). This allows us to ask general questions about birds even if there is no Tweety (or any birds):

\[
\text{bird:can\_walk?}
\]

The second 'problem' with the is-a formalism is that it does not actually suppress variables. This would mean, for example, that there would be no benefit in describing a generic sort program within the formalism. Furthermore it is difficult to see how to use is-a notation to describe a module system.

2.8.2 Hayes' Logic of frames

One of the earliest attempts to formalize frame based notation was [Hayes79]. His approach is quite similar to the is-a style we saw above. Hayes characterizes a frame (which we shall assume to be similar to an object) as a conjunction of properties. For example, if we determine that a house should include a kitchen and a garage etc. then this would be written as the \( \lambda \) expression:

\[
\lambda x. \text{house}(x) \supset \exists y_1, y_2 \text{kitchen}(x, y_1) \land \text{garage}(x, y_2)
\]

In other words if an object is a house then it must also have a kitchen and a garage (and whatever else is required of a house).

In his paper Hayes convincingly casts in logical terms the kinds of inferences which seem common in frame based formalisms. These include the criteriality assumption — if an object has a kitchen and a garage then it must be a house, default reasoning and reflexive — or metalogical — reasoning. He does not discuss inheritance; however, we see no reason why an extension similar to that in the is-a approach could not be applied.

Since we do not explicitly give identifiers to our objects in the class template formalism we cannot directly express the criteriality assumption. The criteriality assumption for houses (say) would be expressed in Hayes' formalism as:

\[
\forall x. \text{house}(x) \equiv (\exists y_1, y_2 \text{kitchen}(x, y_1) \land \text{garage}(x, y_2))
\]

We have used class rules in order to describe inheritance and to compute instances
of relations belonging to a class. We could however use a class rule as a criterion: if we wanted to show that a particular object $G004$ (say) was a house (say) then normally we would write the class rule:

$$G004 \leq \text{house}.$$ 

However, instead of using such a class rule for determining properties of the $G004$ object we could — with a different proof procedure than that of Prolog — use it to test if all the predicates of $\text{houedom}$ hold for $G004$, in which case we might conclude that $G004$ is indeed a house.

One apparently trivial problem with both Hayes' formalism and the $\text{isa}$ formalism is that the same predicate symbol cannot be used to denote different relations for different frames. This implies that all slot names in all frames must be distinct. In the class template formalism this problem does not arise because we focus on the 'frame name' rather than the 'frame instances'. This means that we always have sufficient information to be able to distinguish predicates and hence slot names even when the same symbol is used in different class templates.

The issue is actually more important than Hayes suggests especially in a system which includes inheritance. This is because we use inheritance to describe the relationships between relations in different objects. In this context we may sometimes wish to identify two occurrences of a predicate symbol as being the same but in other circumstances they may represent different relations. It is not enough to relegate the problem to "nothing more than syntactic sugar".

### 2.8.3 The POL system

Another approach to incorporating features of object oriented programming is taken in the POL system [Gallaire'86]. This system incorporates some aspects of the $\text{isa}$ style of objects with some new features; these include a form of object search together with a set of higher-order operators for various forms of object search.

A POL program consists of regular Prolog clauses mixed — at the same syntactic level — together with some special declarations relating to objects and classes. An individual object is declared via an assertion in the $\text{instance}$ relation. As with the $\text{isa}$ formulation objects and classes are identified by constant symbols which denote the elements and sets concerned. Again, as with the $\text{isa}$ approach, before we can discuss the properties of individual objects, it is necessary to define one:
tweety instance penguin.

Here tweety is the object name and penguin is the class identifier. As we shall see, declarations such as these are useful when an object retrieval search is requested. Associated with an object are a set of instance variables. These are represented by a series of assertions where the slot name is the predicate symbol and the object and value its arguments:

\[
\text{age(tweety,3)}.
\]
\[
\text{colour(tweety,blue)}.
\]

This representation makes it easy to access an object's instance variables; however as with Hayes's objects slot names must be unique: two classes of objects should not use the same name for different slots.

Defining and invoking methods is more complicated than defining and accessing instance variables. A method of a class is declared with a with statement:

\[
\text{penguin with X\text{:mode(swim)}}.
\]

The “x” variable is mandatory and fulfills much the same role as the self keyword in the class template notation. In order to invoke an object’s methods we use the special “:” predicate (written in infix):

\[
...,\text{tweety\text{:mode(X)}},...
\]

In [Gallaire'86] great emphasis is placed on the fact that such calls are “fully backtrackable” returning alternative solutions as required. However complications seem to arise — especially with the use of cut in methods — which require more elaborate types of declarations for methods.

The “:” primitive uses the instance axioms to determine the class of the object and then uses the isa axioms in a search of the inheritance hierarchy for a definition of the method. The exact inheritance algorithm used is not clear from [Gallaire'86] except that methods which are ‘nearer’ to the object’s class are tried first. Full multiple inheritance is implemented with alternative paths in the inheritance hierarchy being tried when a given method fails.
§2.8 Related work

As we noted above, there is no specific self keyword in the POL syntax; however every method definition must have a variable which is instantiated to the object which invokes the method. The value of this variable is analogous to the value of the self keyword in a class template program. We can use this 'self' variable to define the equivalent of our mode of travel rule for fast-moving animals:

\[
\text{animal with } X:\text{mode(\text{run})} := X:\text{no\_of\_legs}(2).
\]

\[
\text{animal with } X:\text{mode(\text{gallop})} := X:\text{no\_of\_legs}(4).
\]

When such a method rule is invoked, the variable "X" would be bound to the name of the original object used in the original call. Therefore the sub-goal "X:\text{no\_of\_legs}(2)" would be evaluated with respect to the original object not necessarily with respect to the animal class. By associating a self variable with each method the system ensures that every method has a reference to the self keyword whether it is needed or not.

However, since there is no structural grouping of method declarations, it is likely that this self variable will be used more often than self would be needed in class template notation. There is no concept of a syntactic scope for a method declaration; as a result there is no default way of indicating which objects the conditions in the body of a method declaration would refer to. This is compensated for by referring to the self variable to establish dynamically the scope of a call.

Apart from the normal with declaration of a method there are two other types of method declaration. These are introduced by using withdefault or withdeterministic in place of with in the declaration of a method. These keywords give additional control information to the inheritance algorithm and '::' evaluator.

If a method is declared using withdefault then it will only be invoked if no other method is available. This is intended to assist in programming lemma generation — typically via some kind of 'ask the user' process.

The withdeterministic method declaration is used when a method definition/rule is intended to override other rules. If a rule with this declaration succeeds then no other rule for the same method will be attempted. This allows the programmer to control the inheritance of methods, in particular to override inherited definitions with local ones. This also requires the programmer to specify the inheritance control with every method declaration whereas in class template notation the control of inheritance is declared at the point of inheritance (i.e. in the
class rule) rather than with each method.

One problem with this way of controlling inheritance is the fact that the system must use a different semantics for the cases of a 'normal' object call and an 'object search' call. In the latter case both cuts and withdeterministic declarations have a different interpretation. Cuts must be ignored and withdeterministic declarations must be reinterpreted as something closer to the normal with declarations. This is to ensure that the object search is not terminated too early by the presence of cuts in the bodies of methods in order that other objects with similar methods can be tried.

The object search facility is something of a feature in POL. Much effort has been made to make it a useful addition to the repertoire of object manipulation. Included in this is the suite of higher-order operators — designed along the lines of the setof operator in Prolog — which allow one to perform filtered searches of the set of all known objects. For example,

..., laocwc([bird], X:mode(M), L), ...

would search the instances of the bird class, and produce in L a list of pairs: (X, M) where X is a known bird and M is a mode of travel for it. Primitives such as these build a bridge between object oriented programming and databases.

The POL system provides a lot of the features found in object oriented programming but does not really provide a coherent philosophy of objects; certainly not with respect to logic programming and objects. [Gallaire '86] does not attempt to construct a declarative semantics for POL objects; so in that sense the POL system is an implementational system rather than a descriptive one. On the other hand it is not clear from [Gallaire '86] that POL supports assignment although we believe that it may.

One feature of classical object oriented programming — namely a syntactic construct relating all the information about an object together — is missing completely in POL. Indeed the instance values of individual objects and the methods of classes are more likely to be listed according to the relation involved rather than the object involved.
§2.8 Related work

2.8.4 Zaniolo's objects

One of the earliest attempts at combining logic with object oriented programming was by Zaniolo in [Zan'84]. In that paper Zaniolo shows how a limited form of object oriented programming can be implemented in Prolog. As with class templates, he also uses a translation between object oriented programming and Prolog although the details of the mapping are different. Like class templates, Zaniolo's system does not support objects with state; however he does have some of the other features of object oriented programming.

Our person example would be written as a rule for the with predicate:

\[
\text{person}(\text{Age}) \text{ with } \{
\text{age}(\text{Age}),
\text{likes}(X) :- X:\text{likes}(\text{logic})\}.
\]

The notation for calling an object is the same in Zaniolo's system as it is in ours\(^\text{13}\). An object call using the "::" operator is evaluated by searching the with definitions for one which matches the object — on the left of the "::" call — and then invoking the appropriate clauses contained in the with list.

There is a simple scheme for inheritance hierarchies; so to declare that tom (say) is a particular kind of person we would write:

\[
\text{tom isa person}(33).
\]

or that people are examples of life-forms:

\[
\text{person}(A) \text{ isa life_form}.
\]

It is possible to have several isa declarations in a class; however it is though each of the isa declarations were an overriding class rule: there is no genuine nondeterministic multiple inheritance in the same sense as for class templates. Nor is there any notion of self or super.

Like POL, there is a form of object search. This arises from the fact that the with declarations of a class are also Prolog statements. So, a program can be constructed which uses this database to search for objects which have some

\(^{13}\) It seems that the notation for sending a message to an object is agreed upon by many workers even if nothing else is!
required attributes. However, unlike POL, there is no assistance given to the programmer to make this search easier to specify. This means that object searches must be explicitly constructed by the programmer through a recursive program.

We differ from Zaniolo in two main respects: we are not directly interested in implementing a conventional object oriented programming system in Prolog; rather we are concerned with what objects are. We have built up from first principles a language system which has many of the features of object oriented programming languages but which is also soundly based.

The second way in which we go beyond Zaniolo is in the scope of our system. We believe that the use of broadcasting is unique to our work, for example, as is the nondeterministic multiple inheritance. Broadcasting is especially useful when modelling physical systems: if one models an object as a collection of components then many calculations will involve all the components.

Finally, since Zaniolo does not include any notion of self or super, it is difficult to express rules such as our rule for fast-moving animals. This is because we must be able to refer to the context of a message when 'answering' it.

2.8.5 LOGIN — Logic with inheritance

The LOGIN language [Ait-Kaci & Nasr'86] presents us with another way that we might incorporate objects into logic. In fact LOGIN is not an object oriented programming system in our sense: there is no identified notion of objects, methods and classes for example. Nor is there any attempt to organise programs 'in the large'. However two features unique to LOGIN allow us to describe inheritance 'in the small'.

LOGIN is a typed language; all terms have a type associated with them. Unification between terms involves unifying the associated types of each term as well as unifying the terms themselves. When unifying two terms of different types the types of both must be coercible to the same type; this is only possible if the terms have the same type or if one of them is a sub-type of the other. Type declarations allow the programmer to state that a set of terms of a given type are a sub-type of another type. In this way we can describe some kind of inheritance:
§2.8 Related work

penguin < bird.
bird < animal.

These two sentences state that terms of type penguin are of type bird, and terms of type bird are of type animal; and in consequence penguins are also animals. In fact the symbols penguin, bird and animal here refer to the function symbols of terms. Thus, for example, we know from these two sentences that terms whose function symbol is bird are a sub-type of terms whose function symbol is animal.

The second innovation in LOGIN is to use $\psi$-terms instead of conventional terms. Arguments in a complex $\psi$-term (corresponding to a compound term) are indicated by attribute/value pairs. So, if we wanted to describe a conventional term such as:

\[
\text{tree}(\text{left}_\text{tree}, \text{lab}, \text{right}_\text{tree})
\]

as a $\psi$-term (where left_tree and right_tree are also trees) then we would write:

\[
\text{tree}(\text{left}=>\text{left}_\text{tree};
\text{right}=>\text{right}_\text{tree};
\text{label}=>\text{lab})
\]

The order that we write the sub-terms in a $\psi$-term is not important since the arguments of a $\psi$-term are identified by selector functions rather than by their position.

It should be obvious that there is a one-to-one correspondence between equivalent $\psi$-terms and conventional Herbrand terms. Furthermore $\psi$-terms are an extension of terms to the standard database formulation of atomic formulae where arguments can be represented by attribute/value pairs rather than being indicated by their positions within the formula. Apart from the extra documentation inherent in $\psi$-terms there is also the syntactic convenience of omitting unreferenced arguments and of not having to follow a fixed ordering of arguments. This syntactic convenience is balanced by a slightly more difficult representation of shared variables (where the expected type of the variable must be given) and the fact that circular terms can be described.

However a number of the claims suggested by [Aît-Kaci & Nasr’86] simply do not follow. In particular they claim that using type inference is 'more efficient' than
using standard resolution. This assertion ignores the perfectly straightforward option of performing similar type inferences in a conventional logic programming system at compile-time, prior to the execution of any queries. Furthermore their unification algorithm is somewhat more complex than conventional unification and is less amenable to compilation.

Using the type declarations we can describe a kind of inheritance over terms (and hence objects). However this inheritance is somewhat weaker than the inheritance that we have discussed so far. There is no possibility of overriding or default inheritance. Nor can we express notions which involve the use of self. Therefore we cannot express — in the type language — our rules for fast moving animal for example.

Whilst LOGIN may represent a useful way of incorporating types into logic programming; it also shows that there is a gap between the requirements of object oriented programming and type theory.

2.8.6 Logical objects

Almost in strict contrast to LOGIN, the problem addressed in [Conery’88] is that of representing objects with a mutable internal state. The contrast is enhanced by the fact that [Conery’88] does not even attempt to address issues such as program structure and inheritance.

Classes in [Conery’88] are represented by specially paired clauses called ‘object clauses’. So, for example, a person class which has methods in it for likes and age (say) would be written:

\[
\begin{align*}
\text{person}(A, \text{ID}) & \wedge \text{age}(\text{ID}, A) \leftarrow \text{person}(A, \text{ID}). \\
\text{person}(A, \text{ID}) & \wedge \text{likes}(\text{ID}, X) \leftarrow \text{person}(A, \text{ID}) \wedge \\
& \quad \text{age}(X, Y) \wedge Y < A.
\end{align*}
\]

where person(A, ID) is an ‘object literal’ and age(ID, A) and likes(ID, X) are ‘procedure literals’. (We use a different font to distinguish object literals from procedure literals.)

A query to this program is really a pair of queries which are linked through shared variables. One part of the query involves only procedure literals; the other part of
the query consists of object literals which represent the objects in the system. A complete proof involves solving both — proving the existence of the objects and proving that those objects satisfy a given set of constraints. The two sub-proofs are linked and are performed in parallel; furthermore they may share variables.

If we wanted to find out if a particular person jack (say) liked another person jill (say) then we would use the query:

\[
\leftarrow \text{new\_person}(\text{jack}, 20) \land \text{new\_person}(\text{jill}, 18) \land \\
\ldots \land \text{likes}(\text{jack}, \text{jill}).
\]

The new_person predicate has a special role in the description of a class. Each class must include such a program. Its primary function is to introduce a new object literal into the system. Our new_person program may be defined as:

\[
\text{person}(\text{Name}, \text{Age}) \leftarrow \text{person}(\text{Age}, \text{Name})
\]

From a query such as the one above we can construct a proof which looks like

\[
\leftarrow \text{person}(20, \text{jack}) \land \text{person}(18, \text{jill}) \land \ldots \land \\
\land \text{likes}(\text{jack}, \text{jill})
\]

\[
\leftarrow \text{person}(20, \text{jack}) \land \text{person}(18, \text{jill}) \land \ldots \land \\
\land \text{age}(Y, \text{jill}) \land Y<20
\]

\[
\leftarrow \text{person}(20, \text{jack}) \land \text{person}(18, \text{jill}) \land \ldots \land \\
\land 18<20
\]

After the main proof is completed (which in this case is with the proof that \(18<20\)) the object proof continues to verify that person(20,jack) and person(18,jill) are indeed valid objects. This is done by programmer supplied clauses which define what a valid object is.

It is possible to model state in this system because it is the object literals in the query which represent objects. These may be transmuted by the object clauses. A object clause such as

\[
\text{person}(A, \text{ID}) \land \text{new\_age}(\text{ID},B) \leftarrow \text{person}(B, \text{ID})
\]

'consumes' a person object of age A from the current query and 'replaces' it in
the new query with a person object of age B. The objects themselves are modelled by recursion over the object clauses. Each time a method is invoked a new copy of the object literal (with presumably more up-to-date values for its arguments) is introduced to replace the one consumed by the method.

It is worth pursuing the exact logical status of these ‘object clauses’ a little more deeply. In general a query — consisting of a mixture of object literals and procedure literals — represents a request for a proof of the existence of a set of objects and for the truth of a number of conditions. Since object clauses do have a declarative semantics we should be able to compare an ‘object proof’ with a more conventional one.

Suppose that we have a query of the form:

\[ \leftarrow O_1, \ldots, O_i, P_1, \ldots, P_k \]

where the \( O_i \) are object literals and the \( P_j \) are procedure literals. Suppose that we also have an object clause

\[ o \wedge p \leftarrow b \]

or more strictly

\[ \forall X o \wedge p \leftarrow b \]

where \( o \) is an object literal, \( p \) is a procedure literal, \( b \) is a conjunction possibly involving both types of literal and \( X \) is a vector of the variables occurring in the clause. Since we know that an object clause is logically equivalent to two clauses we write them so:

\[ \forall X p \leftarrow b \]
\[ \forall X o \leftarrow b \]

Now suppose that \( o \) and \( O_i \) unify for some \( i \) in the query, and also that \( p \) and \( P_j \) unify also. Then we might proceed with our query as follows:

\[ \leftarrow (O_1, \ldots, O_{i-1}, O_i P_1, \ldots, P_{j-1}, b', P_{j+1}, \ldots, P_k) \theta_1 \]
\[ \leftarrow (O_1, \ldots, O_{i-1}, b'', O_{i+1}, \ldots, O_i P_1, \ldots, P_{j-1}, b', P_{j+1}, \ldots, P_k) \theta_1 \theta_2 \]

The object clause inference rule involves replacing both parts of the head of the object clause simultaneously: there must be both a procedure literal \emph{and} an object literal in the query which unifies with the procedure head and object head of the object clause. The condition that \( O_i \) and \( P_j \) are replaced simultaneously amounts to
a further constraint that $b'$ and $b''$ are unifiable (with m.g.u. $\emptyset_3$ say). If we apply this substitution to the new query we obtain two sets of literals identified as $b'\emptyset_1\emptyset_2\emptyset_3$ and $b''\emptyset_1\emptyset_2\emptyset_3$ which are identical, and therefore we can factor one set out to get the new query:

$$\langle O_1, \ldots, O_{i-1}, O_{i+1}, \ldots, O_{i}, P_1, \ldots, P_{j-1}, b', P_{j+1}, \ldots, P_k \rangle \emptyset_1\emptyset_2\emptyset_3$$

This new query is the one which is derived from our original query using the object clause inference rule employed by Conery. The effect of it has been to replace two literals in the query at the same time.

Compared with standard resolution this inference is somewhat restricted — it enables literals which are not necessarily logically related to be made identical. (If similar inferences were to be applied to the so-called procedure literals then many recursions would become impossible.) A consequence of this type of inference is that the logic itself is weakened. For example, this system loses a fundamental property of logical proofs — namely that the steps within a proofs can be reordered without changing the validity of the proof. In Conery's system the order of the steps in an object proof is crucial.

Logical Objects gives a semantics to object oriented systems in terms of a modified proof procedure. This allows Conery to state that executions of object programs correspond to logical consequences of theories. However there is no attempt to give systems of evolving objects a declarative semantics as well as a proof theoretic one. The lack of such a declarative reading for objects weakens the argument that Logical Objects are logical.

However, the logical object system does represent an interesting way of representing objects with state. By modifying the proof procedure to allow the modelling of assignment we obtain a simulation system which allows us to simulate state change rather than a purely descriptive formulation. The key issue of whether state-change systems are logical at all, and how one can incorporate changing state with declarative semantics is ignored.

### 2.8.7 Objects as intensions

In [Chen & Warren'88] objects are modelled as intensional variables. Intensional logic is concerned with the logic of possible worlds in general and with time in
Chapter 2: Logic and Objects - the class template language

In particular. An intension is a

"function from sets of possible worlds into values of some domain"

In particular the set of possible worlds relates to the set of worlds along a time-line. An intensional variable has as a value a set of state/value pairs — each state corresponding to a possible world.

Chen & Warren have used this concept to identify objects as intensional values. An object would be a set of state/value pairs representing the history of the object in the various states. In a practical proof procedure the intensional value of an object is only partially known — its past history is known but its future is not. A frame axiom inference rule is used to linearize all the states of all the objects/intensional variables so that any object can be set to have a value in all the historical states in the system.

Chen & Warren provide two operators for manipulating an object: IS : X is a predicate which is satisfied if the object denoted by the intensional variable IS has value X in the ‘current’ state. The := primitive is used to manipulate states: IS := V adds another state to IS with V as the value in the new state.

By using intensional variables a dynamic object modelling system can be constructed. Furthermore the semantics is founded on Montegue’s system of intensional logic [Montegue’73]. However, there are many different proposals for modelling state in the context of logic programming. It is not clear that an appeal to a more sophisticated logic such as intensional logic is required. As with [Conery’88], Chen & Warren have not adequately addressed the issues of program structuring, much less inheritance.

2.8.8 Concurrent logic programming and objects

Shapiro [Shap’83], Kahn[Kahn’86], Davison[Dav’88] and others have looked at the implementation of object oriented programming languages in concurrent logic programming languages such as Concurrent Prolog and Parlog. Shapiro’s work was primarily concerned with the implementation of the operational semantics of object oriented programming languages in Concurrent Prolog although this has since been ‘sugared’ up into a class notation by Kahn and Davison.

This work is interesting for two reasons: it represents one of the few object oriented
programming systems which actually use message passing and which actually do execute non-sequentially, and secondly the object oriented programming notation represents a useful paradigm for using these languages. Perhaps by using such a notation the concurrent logic languages can be ‘honest in their intentions’ once more!

An object in ‘Concurrent object oriented programming’ consists of a continuously executing process. Messages are sent to the object in the form of a stream which the object process consumes and responds to. Object processes may optionally have output streams of messages which are connected to other object processes.

Interestingly, inheritance is also implemented in the same way. An object process only consumes those messages on its input stream which it can directly handle. Any other messages are passed through it onto another exception stream. This exception stream is then connected to another object process which is the super class of the original object. Of course the super object may also pass messages onto its super object and so on. These interconnecting streams are set up whenever an object is created for the first time, which makes it quite expensive to create an object as all its superior objects must also be created.

Implementing inheritance in this way does potentially allow for inheritance links to be dynamic. In our work inheritance is always static; this allows the compiler to resolve inheritance links.

Local state is modelled within an object process by the use of recursion. State variables are held as non-stream arguments to the object process. As an object changes state then the process recurses with new values in these variables. Object processes can be shared (and hence all see the current state of the object) by means of merge processes which connect the object process to all the clients of the object. The merge processes merge the message request streams of each client process into a single stream before passing that message stream to the object. The merge processes have to make sure that the requests are merged in order that each client ‘sees’ the object process in the same state.
2.8.9 Metalogical objects

Kowalski [Kow'82] and Bowen [Bow&Kow'82] have examined the use of meta language to construct programs. The semantic basis of a metalogical system is of course different to a purely object-language system. In particular Kowalski and Bowen have examined the amalgamation of the meta and object languages for constructing programs.

In the amalgamated approach programs are named and constructed in the meta-language and queries over those programs are evaluated in the object language. This is achieved by reflecting the named programs into actual object level clauses and using the standard Prolog (say) evaluator to prove a query. The results of the query are reflected back into the meta-language.

Such combined systems are quite complex; perhaps a little too complex given the fact that in practice programs tend to be quite static. Furthermore there are non-trivial correctness problems associated with constructing programs and evaluating queries relative to them in the same system: one must be sure to avoid paradoxes relating to self referential statements for example.

On the other hand there are some similarities — at least in spirit — between a metalogical approach to objects where an object might be characterized as a set of true sentences and our approach where an object is characterized by sets of relations.

The key difference, it seems, lies lies in the nature of inheritance. In a metalogical system inheritance is viewed as a metalogical process of unioning sets of sentences together. In our relational system inheritance is an object-level process of unioning sets of relations: i.e. the modules of sets of sentences rather than the sentences themselves.

It may be necessary however, to lift our system into a metalogical one — even though that gives rise to a more complex semantics — in order to successfully deal with evolving systems and state change in general.
Chapter 3:

Denotational Graphics

In this chapter we look at a significant application of class template programming, namely the implementation of a graphics sub-language. Apart from being a significant application in its own right it is also a novel approach to graphics itself. We use recursive term structures to denote pictures. Common operations, such as drawing pictures and testing to see if a point is inside a picture, are described using class template programs. The same combination also forms a sound basis for graphically oriented applications.

§ 3.0  Introduction

Graphics and graphics based applications are important primarily because a good graphical interface to an application makes the application much easier to use. However there is a great deal more to graphics than simply being able "to draw pretty pictures" on a screen or on a plotter. The construction of systems with a graphical style of interface is also important and rather harder to achieve. For example, on the Apple Macintosh, icons are used to represent programs and files, and the graphical operations of pointing at an icon with a mouse and clicking the mouse button is interpreted as the 'logical' action of invoking the program represented by the icon. In fact, from the point of view of a user interacting with a complex system, graphical operations are quite intuitive and easy to use and learn.

It follows that any attempt to link the visual world of graphics to the symbolic world of logic programming must go further than simply drawing pictures. Apart from a language in which to describe pictures logically we also need a 'language' in which the interaction between a user and an application can be described.

In this chapter we examine one such language for describing pictures based on a simple term representation. We show how these terms can be manipulated to compute drawing sequences and to perform other graphical operations. We also look at graphics windows and see how they form a suitable basis for constructing a certain class of graphic applications.
§3.0 Introduction

3.0.1 Objects versus command sequences

A key aspect of the way graphics facilities are embedded into programming languages is the sub-language in which pictures are described and manipulated. Programming languages often contain sub-languages, for example arithmetic expressions in Prolog or data pictures in COBOL. Like any other sub-language there is the problem of integrating the graphics sub-language into the host programming language; in this case the host is logic, or more accurately Prolog. The choice of notation for describing pictures affects the ease of integration and the ease of constructing graphical applications. We aim to produce a notation which is natural for both pictures and logic.

There are fundamentally two styles of graphical sub-language: the procedural or command oriented approach and the object oriented approach. A good example of the former is turtle graphics in the LOGO language [Papert'80]. Turtle graphics were expressly designed to be easy to use so that young children could use graphics as an introduction to mathematics and computing. However turtle graphics is heavily oriented towards drawing pictures, in particular line drawings. It is not as strong in describing interactive graphics applications.

Given the preponderance of procedural languages one should not be too surprised that most graphics sub-languages are also procedural. However a procedural/command based model is not appropriate for a symbolic declarative language such as Prolog, although nearly always the low-level graphic primitives, as understood by hardware, are command based.

An alternative approach is the object oriented approach (not to be confused with object oriented programming). In this approach objects in the language are used to denote pictures on a screen or plotter. The relationship between picture terms and the pictures they denote is the same as between terms in general and objects in the world. For example, the term circle((0,0),10) has the same relationship to a picture of a circle as the term “3” has to the number and peter to a person. In each case the term is said to denote the actual object and manipulation of these terms by logic programs is used to represent computations over the objects denoted. In this chapter we explore this denotational approach to graphics and graphics-based applications.
§ 3.1 Denoting pictures by terms

In our graphical 'language' we employ an abstract syntax of pictures. We use compound terms to denote various kinds of pictures, and we describe relations over those terms which can be used to compute the low-level drawing sequence needed to actually draw the picture, or to describe spatial relationships between pictures such as a point occurring 'inside' a picture.

We classify pictures into three kinds, and these are reflected by three kinds of terms: primitive or simple pictures, aggregate pictures and modified pictures.

3.1.1 Simple pictures

A term such as \((20,30)\) can be used to denote the point \((20_x,30_y)\) in some coordinate space. This is the simplest primitive picture. A slightly more complex picture is the straight line segment. This is denoted by the term \(P_1 - P_2\) where \(P_1\) and \(P_2\) are both terms which denote points. One can allow for a connected sequence of points with a construction such as the term \((10,20)-(30,40)-(50,0)-(10,20)\) which denotes the triangular sequence of line segments:

Although in principle (and in practice at the level of graphical display hardware) one can construct arbitrary pictures from these two primitive types it is convenient to have a larger library of such primitive types. For example we would like to have primitive types to denote circles, text strings, rectangles and so on. We can use terms to describe these graphical objects just as we can for points and line segments: \(\text{circle}(C,R)\) denotes the circle with centre at the point \(C\) and with

\[\text{The construction } P_1-P_2-P_3 \text{ is equivalent to a pair of line segments: } P_1-P_2 \text{ and } P_2-P_3. \text{ This is analogous to the mathematical convention that } x<y<z \text{ is equivalent to the conjunction } x<y \& y<z. \text{ The infix hyphen is a left associative operator.}\]
§3.1 Denoting pictures by terms

radius \( R \), \text{text}(P, F, T)\) denotes the textual display of the term \( T \) with origin at the point \( P \) using the font \( F \).

The richness of the library of primitives determines how much effort is required to describe more complex pictures. In MacProlog [Clark et al 1987] we describe a system with 17 different primitive picture types and over 30 picture modifiers. Below we shall see how we can define a primitive picture by means of a program, this allows us to have any number of primitive types of picture.

We intend to use class template programs in this chapter to describe various relations over pictures. For example, in order to describe the various relations which are true of squares, such as the \text{draw} relation or the \text{points_in} relation, we can collect them together into a single \text{sq} class.

\[
\text{sq}(\text{Centre}, \text{Length}):\
\begin{align*}
\text{draw} : & -... & \% \text{describe the draw relation as it relates to squares} \\
\text{pt_in} : & -... & \% \text{describe the points in relation as it relates to square.}
\end{align*}
\]

By using class template notation as opposed to conventional clausal notation we can group the sentences of the various graphic relationships relations according to the type of the picture object rather than according to the relations being defined. Furthermore, the use of inheritance provides a uniform mechanism allowing us to state that “one picture is a special case of another” once and for all.

3.1.2 Computing with pictures

It is interesting but hardly sufficient to have a denotation of pictures by terms. What is also required is a method of computing over these picture terms. We shall look in detail at two such calculations: computing the sequence of low-level commands which when executed by a suitable graphics processor will cause the picture to be drawn; and determining if a given point is ‘inside’ or ‘outside’ a picture.

3.1.2.1 A simple graphics display processor

Since we are constructing a drawing sequence which will cause a picture to be
displayed, we should make some attempt to specify the acceptable command sequence. There are many different varieties of graphics display hardware, so for the sake of simplicity we shall assume a system which can cope with commands represented by terms such as \( m(X, Y) \) to move the drawing pen to the location \((X, Y)\), \( l(X, Y) \) to move the pen while drawing a line, \( p(H, W) \) to change the size of the pen, \( c(C) \) to change the pen's colour and so on. Using these commands, the drawing sequence to draw a red square of sides 20 units centred at the point \((10, 20)\) could be represented by the list:

\[
[c(\text{red}), m(0, 10), l(20, 10), l(20, 30), l(0, 30), l(0, 10)]
\]

Of course in order to actually draw the square this list of commands would have to be interpreted as the appropriate sequence of commands. Furthermore the command letters \( c, m, l \) etc. are in practice more likely to be integers or character escape sequences than letters. However neither of these considerations concerns us at the moment. (Actually a real graphics display processor is quite likely to be more sophisticated than we have assumed, able to understand high level commands such as draw a circle and plot text in various fonts without us having to reduce everything to a sequence of lines and points.)

3.1.2.2 Drawing a simple picture

In order to draw a picture we have to 'translate' the term which represents the picture into the appropriate list of drawing commands; these commands are then executed on a suitable graphics processor. While the actual behaviour invoked by these commands cannot be said to be declarative, (there is no relation which is described by the execution of the commands) there is a relation defined by the graphical compiler.

A key feature of our compiler is that the data (which takes the form of lists and other term structures) is inherently heterogeneous rather than homogeneous: an aggregation of pictures is a list of pictures, each picture can be of any type and itself be a list of pictures and so on.

A conventional Prolog program to construct the drawing sequence for a square might look something like:
§3.1 Denoting pictures by terms

draw(sq((CX,CY),S),
   [m(CX-S/2,CY-S/2),
   l(CX-S/2,CY+S/2),
   l(CX+S/2,CY+S/2),
   l(CX+S/2,CY-S/2),
   l(CX-S/2,CY-S/2)]).

where a list such as [m(CX-S/2,CY-S/2),...,l(CX-S/2,CY-S/2)] represents a sequence of commands to our low-level graphics driver (which might even be implemented in hardware).

As with our \texttt{was} evaluator that we saw in Chapter 2 we have a problem with this \texttt{draw} program in that it is difficult to extend. In order to add a new graphic entity the centralized \texttt{draw} program must be extended in order to add new cases; something which may be hard to do when the source of the program is not available.

A more general approach uses the "univ" primitive = . . to dynamically construct calls to individual programs to do the actual drawing:

draw(Pict,Struct) :-
Pict = . .[Op|Args]2,
Call = . .[Op,Struct|Args],
Call.

Using this version of the \texttt{draw} program a condition of the form

\ldots , \texttt{draw(sq((10,20),20),List)}, \ldots

reduces to

\ldots , \texttt{sq(List,(10,20),20)}, \ldots

A program called \texttt{sq} would be used to compute the list of commands needed to solve this condition:

\footnote{Notice that we cannot simply use the function/predicate variable form in this case since we cannot easily predict the number of arguments to a picture program.}
Apart from the sheer ugliness of this version of the `draw` program, and the fact that in most Prolog systems the `.` primitive is expensive to use, it is still not sufficient in general: if we also require an operation to test for an interior point we cannot use this technique again to implement the predicate 'point inside picture' since we are already using the predicate symbol `sq` to denote the program which draws rectangles.

We can use our class template notation to describe a graphic picture system in an altogether more elegant way. Each graphic entity is described by a class with local definitions of methods for drawing, computing interior points and so on. In particular the `sq` program might look like:

\[
\text{sq}(C,L):\{\begin{align*}
\text{draw}(S):&=\text{b_left-b_right-t_right-t_left-b_right}:\text{draw}(S). \\
\text{b_left}&=((U-L/2,V-L/2)):\text{C}=(U,V) . \\
\text{b_right}&=((U+L/2,V-L/2)):\text{C}=(U,V) . \\
\ldots
\end{align*}\}
\]

Given that we may wish to construct the drawing sequence for a square, this can be represented by the query such as

\[
\{ (S) \mid \text{sq}((10,20),20):\text{draw}(S) \}\]

This shows how a square is drawn in terms of four line segments. The locally defined functions `b_left`, `b_right`, `t_left` and `t_right` return the vertices of the square. In order to draw a square one draws a sequence of line segments

---

3 For the purposes of exposition we do not show the most efficient programs possible, rather we show obviously correct ones.
3.1 Denoting pictures by terms

around these vertices. The line segment class must itself have a drawing method:

\[ P_1-P_2:\{
\text{...}
\text{draw}(S_1\leftrightarrow[P2:x\_coord,P2:y\_coord]):-\]
\[ P1:\text{draw}(S1). \quad \% \text{draw the first point}
\}

In order to draw a single point we simply move the drawing pen to it:

\[ (X,Y):\{
\text{draw([m(X,Y)]).}
\text{x\_coord=X}. \quad \% \text{define function to return X coord}
\text{y\_coord=Y}. \quad \% \text{define function to return Y coord}
\}

Using these programs we can see how solving the query

\[ \{ (S) \mid \text{sq((10,20),20):draw(S)} \}\]

results in computing the appropriate drawing sequence:

\[ \ldots, \text{sq((10,20),20):draw(S)}, \ldots \]
\[ \ldots, (10-20/2,20-20/2)-\ldots-(10-20/2,20-20/2):\text{draw(S)}, \ldots \]
\[ \ldots, (0,10)-(20,10)-(20,30)-(0,30)-(0,10):\text{draw(S)}, \ldots \]
\[ \ldots \ldots \]

finally resulting in the answer:

\[ S=[m(0,10), l(20,10), l(20,30), l(0,30), l(0,10)]\]

Notice that we are using the fact that the term \( P_1-P_2-P_3 \) is equivalent to the term \( (P_1-P_2)_P3 \), and that drawing a line sequence such as \( P_1-P_2 \) leaves the drawing 'pen' at the point \( P_2 \). This is a property of the graphics processor that we are constructing the sequence for, not a feature of the language that we are using to describe a graphical object.

A primitive may inherit its drawing method from another primitive rather than defining the \text{draw} relation directly, as in a circle being a special case of an oval:
circle(C,R) :

% no method for draw in here
...
circle(C,R) <= oval(C,C,R,R).

This inheritance rule states that

"in order to draw (say) a circle of radius \( R \) at the centre \( C \), it is sufficient to draw an oval with centres at \( C \) and with radii \( R \)."

Using a class rule in this way represents a statement of a kind of \textit{graphical equivalence}: a circle is graphically equivalent to a particular variation of oval. We can use this graphical equivalence to separate out the various aspects of objects in a database. For example, suppose that we had a database of stellar information. In this database there would be representations of stars (amongst other things), so we will want to be able to draw stars as well as have more abstract information about them (such as their emission spectra). With a class rule we can show how to draw a star:

\[
\star \text{ star} <= (0,18)-(23,26)-(8,7)-(8,31)-(23,10)-(0,18). 
\]

whilst in a class body describe the other aspects of stardom.

\section*{§ 3.2 More complex pictures}

Some graphic objects are best regarded as modifications of one or more simpler graphic objects. We have seen a trivial example of this with the connected line sequences used to describe squares and so on. The most important class of modifier is the transformational operator. Operators such as translate, rotate and scale are similar to functions which modify their arguments rather than describe new graphical objects. We can denote such a modified graphical object quite simply by using a term to name the transformation. For example the term:

\[
scale(C,X,P)
\]

can be used to denote the picture obtained by scaling the picture denoted by \( P \) by the scaling factor \( X \) about the centre \( C \). The term
There are many common graphical operations which can be seen as modifying pictures, including the traditional operators such as translate, scale, rotate and shear. Moreover modifying other attributes, such as colour or pen style can also be viewed as applying transformations. For example one can describe a triangle with a red outline and a light green interior by the term:

\[
\text{pen\_colour}(\text{red}, \text{fill\_colour}(\text{lgreen}, P_1 - P_2 - P_3 - P_1))
\]

Notice that we have actually applied two transformational operators to the triangle. The facility of applying operators to already transformed pictures makes it easy to construct complex pictures.

### 3.2.1 Aggregate pictures

Often a complex picture contains many separate elements, each of which is a picture in its own right. These are aggregate pictures. An aggregate picture is a set of component pictures. It behaves as one picture from the point of view of transformations applied to the picture. We use a term such as:

\[
\{P_1, ..., P_n\}
\]
to describe the aggregate of pictures $P_1, \ldots, P_n$. Aggregate pictures allow us to view a collection of pictures as a single picture. For example, the graphical description of a table consists of several distinct parts: the four legs and the table top. These individual components together aggregate to form the table itself. Such a table might be described by the term:

$$\{\text{table}_\text{top}, \text{leg}_1, \text{leg}_2, \text{leg}_3, \text{leg}_4\}$$

We can link the graphical description of a table to the other aspects of tableness (sic) through the rule:

$$\text{table} \leq \{\text{table}_\text{top}, \text{leg}_1, \text{leg}_2, \text{leg}_3, \text{leg}_4\}$$

Of course we also require definitions of these components before we could draw or otherwise use such an aggregate picture.

By using this form of term we are using a device similar to the one we used when discussing broadcasting messages in Chapter 2 to implement the appearance of sets. This is not to be confused with a set abstraction. Normally we will be able to distinguish set abstractions because of their particular form, however we may, at times, have to quote picture aggregations to avoid ambiguity.

### 3.2.2 Using complex pictures

Using terms to denote pictures, and in particular transformed pictures, can make certain applications much easier. We illustrate this with a program which automatically generates a tree description of an arbitrary term. For example, the term

$$f(g(h, [i, [j, k]]), l)$$

has a 'generic' tree diagram which looks like:
§3.2 More complex pictures

The program which can generate such a pictorial description of a term is less than a page in length. It also forms the basis for similar types of tree and graph drawing programs. We show the tree drawn/computed sideways because we can assign a fixed height to each leaf node (the height of the font is fixed) and therefore it is easy to compute the height of any tree. A downward pointing tree is also possible of course but slightly more complicated to calculate.

In describing the picture of a term we use three kinds of nodes; a leaf node (for an atomic or simple term) we have the picture: \( \text{leaf} \) denoted by the term \( \text{leaf}(A) \) where \( A \) is the atomic term.

A functor node has the picture: \( \text{denoted by the aggregation \{node(F), A\}} \) where \( F \) is the function symbol and \( A \) is the subtree corresponding to the arguments of the term.

The simple \( \text{picture is used for lists (In this program we ignore the issue of representing list patterns: i.e. lists which are not nil-terminated.)} \)

A call to the \text{generic} function which calculates the generic tree description is typically of the form:

\[ \text{generic(Term,Height)} \]

which returns the graph of \( \text{Term} \) of a given \( \text{Height} \). The \( \text{Height} \) of the graph gives the vertical height of the tree as it will be drawn, it is actually a measure of the number of terminal nodes in the \( \text{Term} \). The main equations for \text{generic} represent a case analysis of the type of terms one might encounter:

The graph which describes a simple or atomic term is simply a leaf:
Chapter 3: Denotational graphics

generic(A,15) = leaf(A):- \% an atomic symbol is a leaf node
    atomic(A).

A composite term (other than a list) has two components in its graph:
\{node(F),List\}, where node(F) is formed from the function symbol F of
the composite term, and List from the list of arguments. The second clause for
generic uses the 'univ' primitive to dismantle the composite term into the
function symbol and list of arguments. The graph for the argument list is formed in
the same way as the graph of a list of terms.

generic(T,H) = \{node(F),generic(A,H)\}:-
    T=..[F|A]. \% a functor is drawn as a node
    \% with a list of subterms

The most complex case of a graph is the list case. The graph of a list of terms
consists of the graphs of each of the elements of the list drawn one on top of the
other. The graph of the list is completed by a series of connecting lines from a
central root point to each of the sub-graphs.

generic(L,Height) = trans(0,VShift,Sub):-
    list(L),
    Sub=generic_list(L,0,Height,Mid),
    Mid = Height/2,
    VShift = -Mid.

The generic_list function computes the sub-graph for each of the elements of
the list, placing each sub-graph immediately below the previous one (by using the
trans operator) and simultaneously computing the description of the lines
between the root and each sub-term. All the sub-graphs are shifted to the right to
accommodate the lines from the root and shifted upwards by half the total height of
the sub-graphs to center the whole sub-tree.
§3.2 More complex pictures

generic_list([T], HSoFar, Height, Mid) =
{trans(25, generic(T, SH), S),
(0, Mid)-(25, E)}:-
Height = HSoFar+SH,
E = HSoFar+(SH/2).

generic_list([H|T], HSoF, Height, Mid) =
{trans(25, E, generic(H, SH)),
(0, Mid)-(25, E),
generic_list(T, HSoF+SH, Height, Mid)}:-
E = HSoF+(SH/2).

This program is an interesting example of the use of the logical variable. The
Midpoint which is referred to in the construction of the lines from the root to each
sub-graph is only known after all the sub-graphs have been computed. During the
entire generic_list calculation it is an unbound variable.

The term which describes the graph of a term can be quite complicated, for example
the graph of the term we saw above is:
If we examine this term we can see that our simple program for constructing generic graphic descriptions is a little redundant in the descriptions it produces, a more careful program could avoid or simplify this redundancy. It also shows that even simple graphics operations can involve manipulating large structures.

Whilst there may be no more than passing interest in the generic program itself it does represent a class of programs which automatically generate graphs. In MacProlog a similar program is used to calculate the call-graph. This is a graphical way of showing the connections in the user's program. By traversing the call-graph the user can traverse his program, showing the source as required.

Another potential application of automatic graph generation is in graphical debugging packages. In this case the graph would represent the trace of a query.
§3.2 More complex pictures

3.2.3 Drawing complex pictures

Drawing an aggregate picture involves computing the drawing sequences for each of the component parts of the picture and appending them together. We can quite simply describe this with the following class program for "{}":

\{L\} : \\
\{ draw(S):- \\
S=\text{draw\_set}(L). \\
\}

\text{draw\_set}((P,L))=S<=>\text{draw\_set}(L):- \\
P:\text{draw}(S). \\
\text{draw\_set}(P)=S:- \\
P:\text{draw}(S). \\
\}

The \text{draw} method for the aggregation label "{}" simply recurses through the list of pictures of its argument and appends the resulting code sequences together. We note that our program above is slightly oversimplified since we have overloaded the use of the "," operator to denote a point as well as an aggregation of pictures. We can resolve any potential ambiguities by restricting \text{draw\_set} slightly:

\{L\} : \\
\{ draw(S):- \\
S=\text{draw\_set}(L). \\
\}

\text{draw\_set}((P,L))=S<=>\text{draw\_set}(L):- \\
\neg (\text{integer}(P) ; \text{integer}(L)), \\
P:\text{draw}(S). \\
\}

3.2.3.1 Pointwise inclusion in an aggregate picture

Apart from drawing pictures another common operation performed on pictures is to determine if a given point occurs in them. This is, for example, used quite heavily in graphics environments where a mouse pointing device is used to select a picture and the system must discover which picture the user has selected.

For simple types of pictures each class program which defines that picture would
have its own method for determining interior points. Usually such a method is quite simple because the primitive itself is simple. For example, to determine if a given point is in a circle one might use the following query and corresponding class program:

\[
\text{circle((0,10), 20):pt_in((5,4))?}
\]

\[
circle(C,R):\{
\text{pt_in(P):-}
\text{P:sub_pt(C)=P1,}
\text{P1:x\_coord}^2 + \text{P1:y\_coord}^2 < R^2. \% \text{from eqn for a circle}
\}
\]

Where the \text{sub\_pt} function defined in the point class subtracts one point from another:

\[
(X,Y) : \{
\text{sub\_pt((U,V)) = (X-U,Y-V).}
\}
\]

In order to determine if a point occurs in an aggregation it is enough if the point occurs in at least one of the component pictures. We can express this quite simply as “find an element of the aggregation which this point occurs in”.

\[
\{L\} : \{
\text{pt\_in(P):-}
\text{E element\_of L,}
\text{E: pt\_in(P).}
\}
\]

\[
\text{E element\_of (E,L) :-}
\text{\neg (integer(E); integer(L)).}
\]

\[
\text{E element\_of (_,L) :-}
\text{E element\_of L.}
\text{E element\_of E.}
\]

\[
\}
\]
3.2.4 *Drawing modified pictures*

We are not restricted to using a single transformation in describing a picture. For example one can stretch a square into a rectangle as well as rotate it, the term

\[
\text{rot}(30^\circ, \text{scale}((0,0),(2,0.5),\text{sq}((0,0),10)))
\]

could denote the picture:

\[
\begin{align*}
\text{rot}(30^\circ,\text{scale}((0,0),(2,0.5),\text{sq}((0,0),10)))
\end{align*}
\]

In fact we have two kinds of modifier: geometric transformations such as *scale*, *translate*, *rotate* & *shear*, and non-geometric transformations which affect the appearance rather than the shape of a picture. For example the colouring modifiers such as *pen\_colour*, *fill\_colour* and *pen\_style* are non-geometric transformations which affect the colour of the outline of the picture, the colour of the interior of the picture and so on.

We can represent the application of geometric transformations by means of a matrix. In order to draw a transformed primitive picture the vertices of the picture are multiplied by this matrix, and the picture is drawn around these transformed vertices. We can represent an arbitrary transformation with a single matrix because matrix multiplication is associative:

\[
M_1 \ast (M_2 \ast T) = (M_1 \ast M_2) \ast T
\]

where \( M_i \) are square matrices and \( T \) is a vector i.e. a point. In this case \( M_1 \) would represent an outer transformation and \( M_2 \) would represent an inner transformation.
The transformation induced by first applying $M_2$ and then $M_1$ can be encapsulated by the single transformation matrix $M_1 \cdot M_2$. In order to compute the drawing sequence of a possibly modified picture one has to take the geometric transformation into account, and we can do this by including the transformation matrix to be applied as an extra argument to \texttt{draw}. The standard query to compute the drawing sequence of a picture $P$ becomes:

$$P:\text{draw}(M,S)$$

where $M$ is a transformation matrix and $S$ is the drawing sequence for $P$.

Each class program for a primitive picture has to apply this matrix to the relevant vertices before returning the drawing sequence or performing other calculations. For example our point class program becomes:

$$\{$$
$$\begin{array}{c}
(X, Y) : \\
\quad \text{draw}(M, [m(U,V)]) :- \\
\quad \quad \text{multiply}(M, (X, Y), (U, V)) .
\end{array}
$$

\}

Where \texttt{multiply} is a matrix multiplication function used here to multiply the logical coordinates by the mapping matrix to get the actual coordinates. The class programs for the various modifiers also contain methods for drawing and point finding. However a typical draw method in the \texttt{rotate} class (say) does not compute any actual drawing commands. Instead the incoming transformation matrix is modified and this modified matrix is applied to the picture argument of the modifier. For example the rotation transformation matrix is

$$\begin{bmatrix}
\cos\theta & \sin\theta & 0 \\
-s\sin\theta & \cos\theta & 0 \\
0 & 0 & 1
\end{bmatrix}$$

which can be represented by the term

$$((\cos(\Theta), \sin(\Theta), 0),$$
$$(-\sin(\Theta), \cos(\Theta), 0),$$
$$(0, 0, 1))$$

The \texttt{rotate} class will have methods for the \texttt{draw} relation, and the \texttt{pt_in} relation. These methods invoke the \texttt{draw} (and \texttt{pt_in}) methods of the
transformed picture, but using a modified transformation matrix obtained by multiplying the existing matrix by the rotation matrix.

\[
\text{rotate}(\Theta, \Pi) : \{
\text{draw}(M, S) :-
\quad \text{multiply}(M, ((\cos(\Theta), \sin(\Theta), 0),
\quad \quad \quad \quad (-\sin(\Theta), \cos(\Theta), 0),
\quad \quad \quad \quad (0, \quad 0, \quad 1)), M_1),
\quad \Pi : \text{draw}(M_1, S).
\}
\]

\[
\text{pt_in}(M, P) :-
\quad \text{multiply}(M, ((\cos(\Theta), \sin(\Theta), 0),
\quad \quad \quad \quad (-\sin(\Theta), \cos(\Theta), 0),
\quad \quad \quad \quad (0, \quad 0, \quad 1)), M_1),
\quad \Pi : \text{pt_in}(M_1, P).
\]

The other geometric transformations can be described using very similar types of programs; the main differences being the transformation matrix to be applied.

Of course, in our original top level query that we used to compute drawing sequences we need to include an initial 'null' transformation matrix; for example to draw a flattened circle surrounded by a square we would use the query

\[
\{ (S) | \text{scale}((0,0),(2,0.5),\{\text{circle}((0,0),10),
\quad \text{sq}((0,0),10)\}) : \text{draw}(((1,0,0),(0,1,0),(0,0,1)), S) \}?
\]
§3.3 Graphics and Applications

In a real system it is not sufficient to be able to describe and draw pictures, it is as important to allow the programmer to quickly and easily construct applications based around them.

Many modern systems employ a desktop metaphor for describing the interaction between the user and the machine. In such a system the screen is divided into a number of different independent areas called windows. Typically windows relate to and are specialized for each application. For example there may be Word Processor windows and spreadsheet windows as well as program edit windows and system control windows. At any one time there may be several different kinds of windows on the screen, depending on the overall task that the user is involved in. The user switches between tasks by marking one of the windows as the 'current' one; this can be easily accomplished through the use of a pointing device such as a mouse.

Underlying all the windows is an overall desktop manager which is responsible for making sure that the various windows are correctly drawn. The user is free to resize and shuffle windows as he pleases as the different activities he is involved in assume different importance to him. Moving windows is quite likely to cause windows which were hidden to become visible and to obscure previously visible windows, in much the same way as pieces of paper overlap on a real desk.

While the desktop manager can organize which windows are where, it is the responsibility of individual applications to ensure that the contents of windows under their control are accurate. This is accomplished by the desktop manager periodically requesting the application to refresh a particular window (after it has become visible for example). We shall see how a Prolog based graphic edit window would respond to such a request.

The graphic edit window, introduced in MacProlog, is a generic window which can be easily specialized to a class of graphics applications. Each graphic edit window has two drawing areas: the tool pane (usually on the left hand side of the window) and the drawing pane. The tool pane represents a 'palette' of graphical activities which can be invoked, and the drawing pane is where application level pictures are actually drawn.
This model of a graphic edit window is based on a class of applications which we might call *picture editors*. A picture editor is used to construct and manipulate pictures. There are many varieties, for example on the Macintosh there are programs such as MacPaint which is a ‘painting’ program where the system simulates the use of a paint brush. Another well known example of a painting program is PaintBox. This program is used by professional animators to construct cartoons for broadcast by television. Other programs such as MacDraft and other CAD programs are really better suited to constructing diagrams and schematics. In both cases tools in the tool palette represent different kinds of pictures which can be drawn: circles, text, free drawing and so on.

A Prolog application, particularly intelligent database applications, could also use a similar approach. The ‘picture’ being drawn is actually a representation of some database and the tools are implements for querying and manipulating the database. Many real databases are as graphical in content as they are textual, for example a database of available flights between cities can be represented on the screen by a map and lines representing available flights between cities. A system which allowed a user to plan an air journey between various cities might have tools to represent different airlines, to select the itinerary and perhaps a route planning device.
In order for a user to plan a route he selects the cities he wishes to go to, and may also enter other constraints to do with time (say). The system responds with an appropriate schedule, which could be displayed graphically superimposed on the map. The use of a map makes some choices easier to make: alternate routings and destinations can be viewed for example.

By extending our denotational view of pictures slightly we can see how a Prolog application can be easily constructed to make use of the graphic edit window style of interface.

Normally, a complete Prolog application presents to the user a fixed (and usually also small) number of 'pre-canned' queries which represent standard ways of using the application. The most obvious way that an application can be linked to a graphic edit window is to associate each of the standard queries in the application with a tool in the window's tool palette.

By selecting the appropriate tool in the tool palette the user of the application is actually selecting one of the standard uses of application. Further input to a query can often be obtained by allowing the user to interact with the picture (selecting cities from the map, for example, rather than by typing city names through the keyboard). The graphical operations of selecting tools and selecting parts of the drawing are interpreted by the Prolog application as selecting a standard query to evaluate and giving further data to the query.

When an answer to a query is of the form of a selection of the underlying database,
such as a particular flight plan, then the answer can often be given as a graphical 'overlay' to the main view of the database, as in the case of a proposed route being drawn on the map of cities for example.

3.3.1 The link between an application and its tools

We now look in a little more detail how a Prolog application can be bound into a graphic edit window. The set of tools associated with a window is in effect a tool aggregation: e.g. \{tool1, tool2, ...\}. This tool aggregation is given by the application programmer as part of specifying a graphic edit window.

Each time 'something happens' in a graphic edit window a query of the form:

\texttt{tool:Query?}

is executed, where Query could be select or click depending on the context, and tool is the name of an active tool in the window's tool aggregation.

The user interacts with a graphics edit window by clicking in the window with a pointer device such as a mouse. The desktop manager interprets a mouse click as a query to the relevant graphical object. For example if a tool icon in the tool palette is clicked on then it is as though the query

\texttt{tool:select?}

were invoked. The intention behind such a query would be to select one of the tools to be the 'current' tool. This tool picture can be highlighted in the tool palette to indicate to the user that this tool is the current tool. This query would normally be paired with a similar query: \texttt{oldtool:deselect?} to indicate the deselection of another tool before selecting the new one.

If the user clicks in the drawing area of the edit window then a different query is invoked:

\texttt{tool:click(\Pi)?}

where \Pi is the point in the window where the mouse cursor was when the mouse button was clicked and tool is the currently selected tool from the palette. Of course if the mouse has more than one button the query would be in the form:
depending on which button was pressed. Extra arguments might also be added such as the state of various meta-keys, such as the shift or control keys and the time that the button was clicked.

The application program can then interpret a query of this form as a standard entry point. For example, in our airline journey planner, if the user has clicked while the city selection tool is active then the query posed to the database may be of the form:

```
select_city: click(Π) ?
```

The class program for `select_city` should have a method for `click` which may use the point Π parameter to determine if the mouse is over a city. If so then the selected city is added to the user's itinerary and the city highlighted in the window in some way. A simple program which did this could be:

```
select_city: {
  .._click(P) :-
    city(C),
    location(C,P),
    add_to_itinerary(C),
    highlight(C).
}
```

Naturally, it is difficult to ascribe declarative semantics to the essentially behavioural activity of selecting pictures and running queries. However the class notation does adapt quite well to the procedural world of user interaction.

3.3.1.1 Drawing and selecting tool icons

Apart from the methods for `click`, `select` and so on, which describe the actual query associated with a tool, we also have to design an icon for the tool. The icon for a tool will be drawn in the tool palette whenever the window needs to be refreshed. The icon is drawn by drawing the tool object itself, i.e. by posing a query of the form
§3.3 Graphics and applications

{(S)|tool:draw(M,S)}?

The reader will recall that the set of tools in a particular window form an aggregation. In order to redraw a window the complete tool aggregation also has to be drawn, using a query of the form of

{(S)|{select_city,...,tool_n}:draw(M,S)}?

The resulting sequence of commands is used to draw the tools in the appropriate part of the window.

Another common activity is selecting tools. If the user has clicked in the tool pane of a graphic edit window then this is interpreted as choosing a tool. In order to be able to determine which tool is selected a query of this form is posed by the desktop manager:

{(T)| T ∈ {select_city,...,tool_n}, T:pt_in(Π)}?

where Π is where the mouse is within the tool pane. Once determined the select method of the new tool is activated, and the old tool is deselected:

..., T:pt_in(Π), current_tool:deselect, T:select

In order for tools to be drawn and selected there have to be definitions for the standard draw and pt_in methods in the class programs for each tool. We could do this directly, but if the tools icon can be represented by a single picture then we can represent a tool's icon through an inheritance rule. For example the icon for select_city might be a circle with the word city under it:

\[ \text{Ocity} \]

since that may be how cities are marked on the airline map. The picture description for this picture is:

\{circle((0,0),5),text((0,10),courier,city)\}

and the inheritance rule which allows us to draw the icon for the select_city tool, and to determine a mouse hit within it could be:

select_city<=\{circle((0,0),5),text((0,10),courier,city)\}
With a single rule we can express both how a tool icon is to be presented in a graphic edit window and also how to select it. Using inheritance rules in this way is analogous to their use for user defined pictures.

3.3.2 Displaying graphic edit windows

In a complete system there may be several graphic edit windows on the user's screen at any one time. Since most windowing systems allow windows to overlap with each other a given window may be partially or completely obscured. If the user brings such a window to the front then the window has to be redisplayed, ie it has to be refreshed.

In order to draw a window the system has to be able to draw the various parts of the window: the tool palette and the drawing area.

In order for the system to be able to refresh the visible area of a graphic edit window it needs to be able to determine what pictures are associated with the window. This can be done in a similar way to the tool palette: by having an aggregation of all the pictures that the programmer has associated with a window. Then, in order to refresh the graphic window, one merely re-draws the tool aggregate and the picture aggregate.

In practice a graphic edit window may also include display controls such as scroll bars. These would allow the user to scroll the drawing area of the window in case the logical picture area is larger than the physical size of the window. The scroll bars are activated by the user and cause the pictures in the drawing area to 'shift', bringing different areas of the drawing plane into view. For example, in our airline planner we would use scroll bars to move the visible part of the map to different areas. The scroll bars can be interpreted as being equivalent to a user defined translation applied to all the pictures in the window. This translation is a constant transformation applied to the complete picture aggregation, though not to the tool aggregation.

In a similar vein to scroll bars some systems allow the user to 'zoom' the view of a window. If the user wishes an overall view of the whole logical picture plane of a window he 'zooms out', to see more detail he 'zooms in'. This user determined zoom can be represented by an implicit outer scale transformation just as scroll bars can be represented by a trans translation.
Thus to actually draw the set of pictures in the window we draw the modified aggregation:

\[ \text{scale}(\text{zoom}, \text{trans}(\text{scroll\_for\_shift}, \text{Aggregate})) \]

Since the desktop manager controls how and when a window is to be refreshed, we can assume that whenever it is to be redrawn it is as though a query of the form:

\[ \{(S)| \text{window}_x: \text{draw}(M, S)\} \]

were posed. The answer to this query allows the screen to be properly refreshed. Just as with an individual tool icon we can represent how to draw a window by an inheritance rule of the form:

\[ \text{window}_x \leq \text{scale}(\text{zoom}, \text{trans}(\text{scroll\_bar}, \text{Aggregate})) \]

This inheritance 'rule' is quite likely to be very dynamic as the user changing the position and zoom factor in the window. The \textit{Aggregate} also reflects the application's output.

The desktop manager can represent the position of a window on a screen by applying a translation to the window, so that the actual query that is made to draw a window is quite likely of the form

\[ \ldots, \text{trans}(\text{P\_window\_location}, \text{window}_x): \text{draw}(M, S), \ldots \]

Other attributes of windows, such as their current size, can also be represented by appropriate transformations.
§3.4 Related work

There has not been a great deal of work on relating graphics to logic programming. In fact until recently few Prolog systems offered any kind of graphics. There are nowadays many Prolog compilers which include some support for graphics. However they are mostly very procedural and lowlevel in their orientation. Amongst the most interesting approaches have been by [Helm and Marriott 1986], and [Julian 1982]. Our approach is effectively an extension of Julian's work, which was itself based on earlier work by the author, combined with class program notation.

3.4.1 Helm and Marriott

In [Helm and Marriott 1986] Helm and Marriott describe a system of describing pictures by rules. Their system is analogous to logic grammars; for example one of their rules:

\[
\text{square}((B,L), \text{Len}) : -
\]
\[
\text{plus}(B, \text{Len}, R),
\]
\[
\text{plus}(L, \text{Len}, T),
\]
\[
\text{line}([[(B,L),(B,R)], \text{red}, \text{solid}]) \&
\]
\[
\text{line}([[(B,R),(T,R)], \text{red}, \text{solid}]) \&
\]
\[
\text{line}([[(T,R),(T,L)], \text{red}, \text{solid}]) \&
\]
\[
\text{line}([[(T,L),(B,L)], \text{red}, \text{solid}])
\]

describes a solid red square. In order to draw such a square, or recognize the square, this rule is interpreted with a special purpose interpreter which presumably is modelled on a conventional Prolog interpreter. The semantics of pictures is based on providing an alternative procedural semantics to an extended form of logic rule. This is similar to the logic grammar formalism except that in this case an alternative procedural semantics is required to describe the effect of drawing pictures on the screen.

This approach is fundamentally different to ours in several aspects. In order to describe complex pictures they introduce new connectives to conventional logic programs: & and {}. In their formalism a complex picture is represented by a conjunction of simpler pictures, whereas we use term composition and lists to describe complex pictures.4
A graphical transformation, such as translate or rotate, becomes a higher order predicate which contains as an argument the atom(s) which describe the transformed picture. This approach leaves many questions unanswered, for example the logical relationship between formulae which describe pictures and logic programming.

Helm and Marriott suggest by executing their rules in a bottom up approach they can recognise pictures as well as generate them. This is analogous to the use of logic grammar formalisms to generate as well as recognize strings. However we feel that in practice this is of limited utility since more powerful matching techniques are needed for recognizing pictures in digitized photographs (say).

Finally by concentrating on the problem of drawing (and recognizing) pictures they ignore the other commonly used graphical relationships between pictures such as the spatial relationships of being 'inside' or 'above' a picture. As we have seen these other relationships are important in the construction of graphics applications.

§3.5 Summary

In this work we have separated out the denotational aspects of graphics from the drawing and behavioural aspects. We use terms to denote graphic entities such as circles and squares. We also use class programs to describe various ways of using these graphic entities. We denote transformed pictures by making the picture term an argument to the transformation factor. This represents a compact yet powerful notation for describing pictures.

Apart from drawing a graphic object, or rather 'compiling' it into a sequence of a graphics driver commands, one can perform other graphic operations such as determining if a point is inside a picture, or whether two objects overlap. The same mechanism is used to describe how a finished graphics application is constructed in terms of tools and graphic edit windows.

We have used class template notation to describe the various relations denoted by graphics. This formalism allows the programmer to concisely express the essentially object oriented view of denotational graphics.

Of course it could be argued that we have relied on Class programs and the connectives introduced therein. However our use of Class programs is primarily for reasons of convenience: it does not affect the underlying semantics of terms denoting pictures.
Chapter 4
Semantics of equations

In this chapter and in the next chapter we look in some detail at the semantics of our variety of logic programming. We have introduced two types of extension to logic programming: the first introduces equations, functions and expressions into logic programs. Object oriented programming which is the second type of extension we shall look at in more detail in Chapter 5.

Overall our aim is to show that our extensions are conservative. I.e. they do not affect or rely on modifications of the standard logic programming semantics. The reason for approaching it this way is both to minimize our work in establishing a new semantics and to minimize the reader's effort since he does not need to learn a new model theory etc.

However our extensions do have a certain independent semantics from standard logic programming. This is because the language of equations (and objects) introduces concepts and features which are not needed or explained in the basic LP semantics of relations. Thus we may be able to map our language into logic programming (even onto) but we are nevertheless introducing new(ish) features which have their own characteristics. In the case of equations we are introducing equations (which are actually in the same form as standard clauses), expressions and especially higher order features such as \( \lambda \)-expressions and set abstractions.

§4.1 The logic of functions

There is a simple mathematical relationship between relations and functions, namely that a function is a set of pairs (i.e. a relation) with an extra uniqueness condition:

\[
\text{"A set of pairs } F \text{ is a function iff whenever } (X, Y) \in F \text{ and } (X, Z) \in F \text{ then } Y = Z."\\
\]

Furthermore a relation can be viewed as a function from tuples to the finite set \{true, false\}. This latter interpretation of relations does not adequately relate to their use in logic programming where we routinely use these functions in the inverse sense: from the value \text{true} to those tuples which satisfy the predicate.
In giving a sound semantics to a combined functional and relational system we would aim to capture the semantics of both relations and functions in as natural a way as possible. In [Emden & Yukawa'87] various possible ways of interpreting equations are considered. These divide into two approaches: the interpretational approach and the compilational approach.

In the former approach an interpreter for expressions is constructed which augmented by the explicit equations is used to reduce expressions into a canonical form which can then be unified with other terms in the conventional logic programming sense. Narrowing [Goguen & Meseguer '84] can be seen as a particular flavour of this approach. This interpreter effectively gives a proof theoretic or procedural interpretation to equations.

In the compilational approach, expressions are represented by variables together with extra conditions in queries and in the bodies of clauses. The extra conditions act as constraints on the variables such that whenever the constraints are satisfied the variable will have as value the canonical form of the original expression.

For example, suppose that we have the condition with embedded expression:

\[ ..., \text{index}(E, L, I+1, J), ... \]

Such a condition is compiled into the pair of conditions:

\[ ..., +^*(I, I, X), \text{index}(E, L, X, J), ... \]

With \( X \) being a variable that does not appear elsewhere in the clause in which this condition is embedded. The definition of the relation \( +^* \) is such that

\[ +^*(X, Y, Z) \iff X+Y=Z \]

Sentences such as this one linking the \( +^* \) relation to the \( + \) function are called definitional sentences in [Emden & Yukawa'87]. This kind of interpretation is more declarative than that implied in the interpretational approach; it describes functions in terms of their defining relations.
Another advantage of the compilational approach is that it is possible to use a standard unification algorithm in a standard Prolog language implementation to achieve the effect of expression evaluation. In systems which rely on interpreting expressions it is necessary to modify the standard unification algorithm. This can make compiling unification more difficult and less efficient.

4.1.1 Functions, Terms and Canonical forms

In standard functional programming, evaluating expressions is viewed as a process of reducing expressions in order to compute canonical values. An expression is viewed as being reducible if there is an equation whose left hand side matches (unifies) with a sub-term of the expression. Such an expression is reduced by replacing the matched sub-term by the corresponding right hand side of the equation. The irreducible canonical representation of a term is viewed as the value of the expression.

Such a term rewriting system functions (sic) by using the system of equations in a left to right manner: by matching the left hand sides of the equations and replacing matching terms by the right hand side. This is an inherently operational use of the declarative information in an equation: a statement that two terms are equal does not mean that they are only equal when viewed from left to right! In attempting to combine logic and functional programming it is necessary to relate the declarative nature of equations (which state which terms are equal) with the operational use of equations for evaluating expressions.

The observant reader will note that our conditional equalities are in the standard form of clauses; it is perfectly possible to regard an equation as a clause about the = relation. If we can add to, or infer from, the programmer supplied (and system defined) equations the axioms:

\[ x=x. \]
\[ x=y \implies y=x. \]
\[ x=y \implies x=z, \ y=z. \]

Together with a substitutivity axiom:

\[ F(x) \land x=y \Rightarrow F(x/y) \]

\[ x=y \]
where $F[x]$ denotes a formula which mentions $x$ and $F[x/y]$ denotes replacing all occurrences of $x$ by $y$ in $F$, then the resulting $=$ relation is an equivalence relation, furthermore it is a congruence relation.

With any equivalence relation there is a set of equivalence classes associated with it: each equivalence class contains all the terms which are equal according to the defining relation. For example, given the expression $[1, 2, 3]<>[]$ we can construct a set of terms all of which fall into the same equivalence class as this expression:

$$
\{ [1,2,3]<>[], [1,2]<>[3], [1]<>[2,3], []<>[1,2,3], [1,2,3], []<>[1,2,3], \ldots, \text{rev}([3,2,1]), \ldots \}
$$

Each equivalence class contains all the terms which are equivalent under the relation induced by the $=$ axioms. Of course as this example shows, equivalence classes are often infinite in size.

Equivalence relations are important because they allow us to construct other relations: Given an equivalence relation $E$, and another relation $R$, we can construct the relational quotient $R/E$. Where $R$ is a relation over individual terms $R/E$ is a relation over sets, equivalence classes in particular. The definition of $R/E$ is obtained as follows:

"a tuple $(T_1, T_2, \ldots, T_n) \in R/E$, where each $T_i$ is an equivalence class of $E$ $\iff$ for each $t_i \in T_i$ the tuple $(t_1, t_2, \ldots, t_n) \in R"$

Informally, the quotient relation $R/E$ is a relation which reflects the relation $R$ but whose tuples consist of equivalence classes of $E$ rather than individual terms.

Given a set of clauses $P$ (not including any axioms for the predicate symbol $=$) and a set of equations $E$ (as defined through axioms for the predicate symbol $=$), we can construct a quotient program $P/E$ by a direct extension of the definition above.

We claim that by allowing this form of equality in our programs we are actually interested in the relational quotients of programs rather than the standard relational model of the programs. However we do not need to reconstruct logic programming in terms of relational quotients as we see next.
§4.1 The logic of functions

It is a property of congruence relations that in order to determine if a tuple \((T_1, T_2, \ldots, T_n)\) is in the relation \(R/E\) it is sufficient to show that the tuple \((t_1, t_2, \ldots, t_n)\) is in \(R\), where each \(t_j\) is any term in the corresponding equivalence class \(T_j\). I.e. we do not have to show that \((t_1, t_2, \ldots, t_n)\) is in \(R\) for every possible member of the \(T_j\) equivalence classes. In particular one may identify a particular member, the representative or canonical member, of a given equivalence class and use that member when trying to determine tuples of \(R/E\). Thus we can formally justify the use of canonical terms in rewriting systems: any term in an equivalence class can be used as the representative element.

For some, well behaved sets of equations, each equivalence class of an expression contains an element which contains no symbol which appears as the principal function symbol on the left hand side of an equation. Such a term is obviously easily recognizable and forms a good candidate as the canonical representative of the equivalence class. We call such a term the value of the expression and evaluation of an expression becomes computing this canonical member of the equivalence class of the expression. This is why \([1, 2, 3]\) is the 'value' of the expression \([1, 2] <\rangle 3\] even though they are both equal according to the system of equations. There should be at most one of these canonical terms if the equations define functions.

If the equivalence class of an expression contains more than one canonical term then the 'function' is multi-valued, and therefore is not really a function at all but is a relation. If the equivalence class of an expression contains no canonical terms then the function is partial: there is no value for the expression.

A reducible expression is a term of the form \(f(t_1, \ldots, t_n)\) where there is an equation of the form:

\[
f(a_1, \ldots, a_n) = t : - B
\]

or where one or more of \(t_1\) are reducible. Note that it is not required for all of the \(t_1/a_1\) pairs to unify for \(f(a_1, \ldots, a_n)\) to be reducible.

An irreducible expression is a term of the form \(f(t_1, \ldots, t_n)\) where there is no equation for \(f\) and furthermore \(t_1\) are all also irreducible. A canonical term is
irreducible.

An expression has a canonical form if there is a derivation starting from the goal:

\[ X = E? \]

where \( X \) is a new variable not occurring in \( E \), using only the axioms in the program (including the axioms for the "\( = \)" relation, and the axioms for reflexivity and transitivity of "\( = \)") ending in a goal of the form:

\[ X = C? \]

where \( C \) is irreducible. This definition of canonical form would be equivalent to canonical representative provided that we can show that "\( = \)" is an equivalence relation, i.e. provided that "\( = \)" is symmetric.

With the existing formulation of the axioms for the \( = \) equivalence relation (reflexivity, symmetry and transitivity) represented as object level axioms along with the programmer's axioms it is quite easy for an evaluator to enter useless loops during execution. This is especially true of the symmetry axiom. However we can use the properties of reducibility and canonicality to establish symmetry as a theorem.

Suppose that we have the fact that two expressions \( f \) and \( g \) are in the "\( = \)" relation: \( f = g \); then since \( g \) has a canonical form \( r \) we know (by definition) that \( g = r \) it follows (from transitivity) that \( f = r \). If it was the case that \( g \neq f \) then since each expression has exactly one canonical representative then by transitivity it would follow that \( g \neq r \) which is a contradiction.

Notice that if an expression has no canonical form then symmetry may break down: if we have \( f = g \), but in the case that \( g \) is not canonical and no equation for \( g \) applies there may not exist a term \( h \) such that \( g = h \), let alone \( h = f \).

In summary, when we augment a set of clauses by a set of equations which have the property that any expression has exactly one canonical form then the relation induced is an equivalence relation. We can use the fact that it is an equivalence relation together with the substitutivity property to prove that two terms are equal iff they have the same canonical representative. We next look at a simple program
which can be used to compute the canonical representative of an equivalence class.

4.1.2 A simple evaluator for canon

The skeleton Prolog program below, called canon, computes the canonical form of expressions given a set of equations. The predicate canon($E$, $T$) is true if $T$ is a canonical member of the equivalence class induced by the expression $E$.

\[
\begin{align*}
\text{canon}(F, T) & : - \\
\text{reducible}(F), & \quad \% \text{a complex term} \\
F &= ..[Fu|A], & \quad \% \text{is the term reducible?} \\
\text{canon_list}(A, CA), & \\
S &= ..[Fu|CA], & \quad \% \text{rebuild from reduced arguments} \\
S &= G, & \quad \% \text{use one of the equations}^2 \\
\text{canon}(G, T). & \quad \% \text{invoke transitive closure}
\end{align*}
\]

Irreducible terms are already canonical:
\[
\begin{align*}
\text{canon}(T, T) & : - \\
\text{not reducible}(T). &
\end{align*}
\]

\[
\begin{align*}
\text{canon_list}([], []). & \\
\text{canon_list}([S|L], [CS|CL]) & :- \\
\text{canon}(S, CS), & \\
\text{canon_list}(L, CL). &
\end{align*}
\]

The canonicalizer (sic) program looks for ways of using the equations defined by the user to rewrite progressively — using transitivity — reducible terms into irreducible terms; recursively applying itself to arguments of expressions. If an expression has no canonical form then this process will either fail or loop indefinitely and the expression has no value.

The canon program can be viewed as a specification for part of a new unification

---

1. The $=$, primitive (pronounced 'univ') which is in most Prolog systems can be viewed as being defined by a finite axiom schema; for every function symbol and constant symbol there is a tuple in the $=,$ relation of the form:

\[f(X_1, \ldots, X_n) = .. [f, X_1, \ldots, X_n]\]

2. Not to be confused with equality defined via the single clause

\[X = X.\]
algorithm. In order to unify two expressions we first find their canonical representative and then unify these in the normal manner. We can produce a modified version of the unification algorithm by embedding canon in a standard unification procedure at the appropriate points. This implements the interpretational view of equations; this modified form of unification is often called narrowing [Goguen & Meseguer'84].

4.1.3 The effect of evaluation order

The order of evaluation of arguments of expressions is determined by the evaluation order in the canon_list program and in the (*) clause for canon itself. A strict Prolog left-right execution of the conditions in these clauses gives us the equivalent of strict or call-by-value evaluation. This is because we would be fully evaluating the arguments of an expression before attempting to use any equations.

In a system with dataflow coroutining, the evaluation — in canon_list — of the arguments of an expression could proceed in parallel, and even with the evaluation of the function (i.e., with any computations arising from the conditions in the programmer supplied = clauses).

The data dependencies on which dataflow coroutining is based would be those which are implicit in the expression notation: the results of sub-expressions flow in as the arguments of the higher expressions. (A purely random ordering might allow the higher level expressions to guess an answer of the lower level expressions.)

There is some correspondence between the concepts of strict/normal ordering in functional languages and left-to-right/dataflow coroutining in logic programming systems. Since logic programming languages such as Prolog have a relational syntax which is apparently neutral to the order of evaluation, the programmer often has to make a separate declaration relating to the desired control flow in his programs, whereas advantage can be taken of the expression notation to give control guidance as well as simply suppressing intermediate variables.
4.1.4 Compiling expressions

In the compilational approach we make use of the definitional sentences implicit in the use of equations to transform programs containing expressions into normal clauses which do not, and therefore we can avoid having to change the unification algorithm and the corresponding resolution rule.

For example consider the definition of palindrome in the program:

\[
\text{palin}(L):= L \text{ id } \text{rev}(L). \quad \% \text{ L is palindromic if its own reverse}
\]

\[
\text{rev}([])=[].
\]

\[
\text{rev}([E|L])=\text{rev}(L)[E].
\]

\[
[]<X=X.
\]

\[
[E|X]<Y=[E|X<>Y].
\]

\[
X \text{ id } X.
\]

in which we have the reducible term \(\text{rev}(L)\). We can rewrite the \text{id} condition to an equivalent conjunction:

\[
..., \text{rev}^*(L,Y), L \text{ id } Y, ...
\]

where \(\text{rev}^*\) is a special predicate symbol which is associated with the \text{rev} function symbol by means of the definitional sentence:

\[
\text{rev}^*(X,Y) \iff \text{rev}(X)=Y
\]

We can systematically replace all reducible terms in the text of a program by new variables, adding extra constraints to the clauses and equations in which the required value of the variable is specified.

For example, consider the transformation of one of the \text{rev} equations, where in each step the underlined sub-expression is rewritten and new conditions are added from the corresponding definitional sentence:

\[
\text{rev}([E|L]) = \text{rev}(L)[E]
\]

\[
= X[E] \iff \text{rev}^*(L,X)
\]
Equations can also be transformed into clauses about the defining predicate, again by invoking the definitional sentences. For each equation of the form

\[ f(t_1, \ldots, t_n) = G : - B. \]

and definitional sentence

\[ f^*(X_1, \ldots, X_n, R) \equiv f(X_1, \ldots, X_n) = R \]

we generate the clause

\[ f^*(t_1, \ldots, t_n, G^*):- B^* \]

where \( G^* \) and \( B^* \) are themselves the results of transforming \( G \) and \( B \). This last condition ensures that transitive closure of functional expressions is maintained. If we complete the transformation of \( \text{rev} \) above we get a program which looks rather like the standard formulation of naive reverse in Prolog:

\[
\text{rev}^*([], []). \\
\text{rev}^*([E|L], Y) :- \text{rev}^*(L, X), \text{<>}^*(X, [E], Y)
\]

A skeleton of a Prolog program that describes this transformation is \text{tr_axiom}:
§4.1 The logic of functions

\texttt{tr\_term(T,S,(GL,G)):-}
\hspace{1cm} % transform a term
\texttt{\hspace{1cm} \hspace{1cm} T=..[F|A],} \hspace{1cm} % break it up
\texttt{\hspace{1cm} \hspace{1cm} reducible(F),} \hspace{1cm} % is this a reducible term?
\texttt{\hspace{1cm} \hspace{1cm} tr\_list(A,B,GL),} \hspace{1cm} % transform the args of the term
\texttt{\hspace{1cm} \hspace{1cm} append(B,[S],B1),} \hspace{1cm} % form a new argument list
\texttt{\hspace{1cm} \hspace{1cm} G=..[F|B1].} \hspace{1cm} % form an auxiliary goal

\texttt{tr\_term(T,S,GL):-}
\hspace{1cm} % case where term is not reducible
\texttt{\hspace{1cm} \hspace{1cm} T=..[F|A],} \hspace{1cm} % we have to transform the args
\texttt{\hspace{1cm} \hspace{1cm} not\ reducible(F),} \hspace{1cm}
\texttt{\hspace{1cm} \hspace{1cm} tr\_list(A,B,GL),} \hspace{1cm}
\texttt{\hspace{1cm} \hspace{1cm} S=..[F|B].} \hspace{1cm}

\texttt{tr\_list([],[],true).} \hspace{1cm} % transform a list of arguments
\texttt{tr\_list([T|L],[S|K],(G,GL)):-}
\texttt{\hspace{1cm} \hspace{1cm} tr\_term(T,S,G),} \hspace{1cm}
\texttt{\hspace{1cm} \hspace{1cm} tr\_list(L,K,GL).} \hspace{1cm}

This transformation program could be used to extend a Prolog compiler in order to handle equations and expressions by first filtering axioms through \texttt{tr\_axiom} and compiling the result as normal clauses. This is compared to the use of the \texttt{canon} program which would be used to extend the unification algorithm to handle reducible terms dynamically.

### 4.1.5 Quoted expressions and evaluation

The exact interpretation of ` is quite subtle; consider a first approximation to a definition of ` via the equation:

\texttt{`X=X.}

An English reading of this this equation may be:

"any quoted expression is equal to the expression which is quoted."

However, this is not sufficient since the = relation is transitive, and therefore under this equation we would get `2+3=2+3=5. This is not the desired intention of
quote. The correct interpretation has to be expressed by an extra case in the canon program (and the corresponding case in the tr_axiom program):

\[
\text{canon}(\texttt{`}X, X). 
\]

By defining \(\texttt{`}\) in this way we are actually short-circuiting the transitive closure axiom. The possibility of having quoted terms is an extra complication especially for systems based of the narrowing principle. The (already extended) unification algorithm must now be able to recognise the quote symbol as well as other function symbols. Moreover the system must be careful about when quotes can be removed: if they are removed too early then in subsequent reductions the quoting effect may be unintentionally lost. All this represents further complications in an already complex narrowing algorithm.

§4.2 The logic of \(\lambda\)'s and Set Abstractions

It should be clear that there is a correspondence between \(\lambda\)-expressions and set abstractions. We can formalize this by using \(\lambda\) definitional sentences which form an analogue of the first order definitional sentence:

\[
\{(t_1, \ldots, t_n, t_{n+1}) \mid C\}(X_1, \ldots, X_n, R) \iff \\
(\text{lambda}(t_1, \ldots, t_n) \cdot t_{n+1} : - C)(X_1, \ldots, X_n) = R
\]

We will use this form of \(\lambda\)-definition sentence to transform programs with \(\lambda\)-expressions into their relational equivalents using set abstractions. In this way we reduce the question of understanding \(\lambda\)'s to that of understanding set abstractions. In this section we show how to map a program with embedded set abstractions into a first order program.

However, before we can give an adequate treatment of set abstractions we must look at the predicate variable.
4.2 The logic of λ's and Set Abstractions

4.2.1 Applying predicates to tuples

We have seen a number of examples of conditions variously of the form:

\[ \ldots, P(t_1, \ldots, t_n), \ldots \text{ and } \ldots, (t_1, \ldots, t_n) \in P, \ldots \]

where \( P \) is a variable. Since the variable \( P \) is quantified over predicate symbols and set abstractions (all of which serve to identify relations) we have to justify this form of predication which is not obviously a first order formula. A similar problem arises with function variables. However we can reduce the problem to just that of predicate variables by relying on the mapping from expression to terms to map all function variables to conditions involving predicate variables.

Our intended use of these higher order variables is strictly limited. In particular we restrict ourselves to passing functional/relation values between arguments and to applying them to an argument tuple. We do not intend to be able to sensibly answer questions such as: "Are two relations equal?" by unifying two or more relations as this would require higher order unification. (We can however legitimately ask a related question: "Are the tuples of two relations the same?") This is a perfectly normal first order use. Of course such a query may not terminate for infinite relations.

Higher order unification is used in \( \lambda \)Prolog [Miller & Nadathur'86,88], where the ability to unify two functions finds a good application in program representation and transformation [Miller & Hannan'88]. However as Miller and Nadathur point out higher order unification is much more complex than normal unification and in general is undecidable. This means that any procedural interpretation of \( \lambda \)Prolog has a difficult theoretical foundation; furthermore (as they admit also) it is difficult to see how an efficient implementation of higher order unification can be built.

Given our restricted intentions with respect to higher order unification we must show how the uses of higher order constructs that we do allow are interpreted. In particular we must see how to map occurrences of predicate variables (and function variables) into a first order form. We do this by extending our abstract syntax to an abstract syntax for clauses as well as terms.

As for the abstract function case we assume that there is a single multi-adic "\( << \)"
Chapter 4: Semantics of equational programming

predicate, and each clause is actually about this symbol. So a clause of the form

\[ p(t_1, \ldots, t_n) : \neg g_1, \ldots, g_k \]

might be represented by the following abstract clause:

\[ \langle p, t_1^a \ldots t_n^a \rangle \leftarrow g_1^a \land \ldots \land g_k^a \]

where \( t_j^a \) and \( g_j^a \) are the abstract representations of \( t_j \) and \( g_j \) respectively. In this scheme a predication with a variable for a predicate symbol such as:

\[ p(e_1, \ldots, e_m) \]

is represented as the abstract tuple

\[ \langle p, e_1^a \ldots e_m^a \rangle \]

in an exactly analogous manner to the abstract view of terms that we saw earlier. Of course since the first element of a tuple can be any abstract term there is no cause for alarm when that term is a variable. The proof procedure for our abstract program is slightly different to that of conventional logic programming since in a resolution step we may be unifying abstract predicate symbols as well as arguments to the predication.

This model of terms does not need to affect the implementation of the language in any way. The normal predicate indexing that a Prolog compiler uses to gain fast access to the clauses for a given predicate symbol can still be used. Furthermore the standard optimization techniques such as tail recursion optimization can still be applied albeit in a slightly modified form.

One situation that can arise however, is that the system may attempt to evaluate a condition whose 'predicate' is still a variable at the point of the call. One possible response would be for the system to attempt all the clauses for all the predicate symbols in the program whose arities match. Alternatively, if the system has built into it the ability to delay the execution of some goals, the system could delay such a goal until the predicate variable becomes bound. An equally legitimate response for a strict Prolog-style system would be to report an error.
§4.2 The logic of λ's and Set Abstractions

Notice that by having a uniform abstract syntax with a single predicate symbol we have not substantially affected the semantics of our programs. It is trivial to show that any relation defined by the original concrete program is a sub-relation of the abstract program, and that any computed answer of the abstract program is the abstract representation of a corresponding actual answer of the actual program.

4.2.2 Giving away names

The next step in understanding set abstractions in a first order context involves giving names to nameless entities. (Recall that set abstractions are relations with no name.) The way to do this was first identified in [Warren'82].

Since we know where every set abstraction is — by examining the text of the program — we can replace each one with a new, unique but otherwise normal, symbol together with a set of defining axioms for the new symbol.

For example, suppose that the first set abstraction that we come across is in the goal:

...,\text{map}\left(\{(X, Y) \mid Y=X+1\}, [1, 2, 3], L\right), ...

we replace the set abstraction by a symbol $\lambda_0$ to get the new goal:

...,\text{map}\left('\lambda_0', [1, 2, 3], L\right), ...

We replace the set abstraction by a symbol $\lambda_0$ to get the new goal:

...,\text{map}\left('\lambda_0', [1, 2, 3], L\right), ...

together with a defining clause for $\lambda_0$ whose body consists of the body of the set abstraction:

'$\lambda_0'(X, Y) :- Y=X+1.$

The value of the (**) goal, in the sense of the relation defined by it, is unchanged by this transformation. Notice that if a set abstraction is directly applied to arguments in a call then it can be expanded in place and a rule of this form is not necessary.
4.2.3 Set abstractions with free variables

There are, in fact, two types of set abstractions: the closed form, where there are no free variables, and those which have at least one free variable. The mapping introduced above is adequate for closed set abstractions, however the treatment of set abstractions with free variables is a little more complex.

We define a free variable to be any variable which occurs in a set abstraction (or lambda expression) which also occurs elsewhere in the clause (or equation) in which the set abstraction is embedded. All other variables which occur in the set abstraction are considered to be bound by the set abstraction.

If a set abstraction has any free variables in it then instead of identifying it by a constant symbol we use a compound term which includes as arguments all the free variables in the set abstraction; for example in the goal (assuming that $R$ is free):

\[ \text{map (} \{(X, Y) \mid Y = R + X\}, [1, 2, 3], L) \]

we would replace the set abstraction by the term \$\text{lambdal '(R)\} to get the goal:

\[ \text{map (} \$\text{lambdal '(R), [1, 2, 3], L) \]

and as before we define a new program for the \$\text{lambdal symbol. However the head of the clause is somewhat unusual:

\$\text{lambdal '(R)(X, Y) :- Y = R + X.\]

The abstract form of this clause shows more clearly the exact nature of the clause:

\[ \text{««'\$\text{lambdal' R» X Y» ← «+ R X Y»}\]

Recall that in the abstract clause notation the predicate symbol is simply a term (usually a constant) which is distinguished by virtue of the fact that it is the first element of the predicate tuple. In this clause we have a more complex term where normally we would have a constant but otherwise this is a normal abstract clause.

In general, if a set abstraction of the form:
§4.2 The logic of \( \lambda \)'s and Set Abstractions

\[ \{(t_1, \ldots, t_n) \mid C\} \]

has free variables \( X_1, \ldots, X_k \) in it, then it is replaced in the text by a complex term of the form:

\[ \text{'$\lambda x x'$}(X_1, \ldots, X_k) \]

and the corresponding defining axiom for \( \text{'$\lambda x x'$} \) which is of the form:

\[ \text{'$\lambda x x'$}(X_1, \ldots, X_k)(t_1, \ldots, t_n) :- C. \]

or in abstract notation we would have the clause:

\[ \langle \langle \text{'$\lambda x x'$} \ X_1 \ldots X_k \rangle t_1^a \ldots t_n^a \rangle \leftarrow C^a \]

This gives the correct interpretation of free variables, in the sense that they take on their value at the time that the set abstraction is evaluated, not when it is applied.\(^3\)

4.2.4 Union set abstractions

If a union of set abstractions is to be given a name then the whole union is replaced by a single generated symbol (parameterized by any free variables in the whole of the union); we then get two (or more) rules induced for the newly defined symbol. For example in this goal we have a union set abstraction which identifies a particular Key:

\[
\ldots, \text{map} \left( \{(X, 1) \mid \text{Key on } X\} \mid \right.
\{(X, 0) \mid \text{not Key on } X\},
\left. \begin{array}{cccc}
[a, b], & [c, \text{Key}], & [d] \end{array}, L\right), \ldots
\]

As before we replace the set abstraction with a new symbol and transform the goal into one using this new symbol:

---

\(^3\) In order to allow efficient (i.e. constant time) access to the clauses for the normal case of a clause a Prolog style implementation may well compile \text{'$\lambda x x'$} clauses in a different way to normal clauses; perhaps in the same style as closures are implemented in functional languages.
together with these rules for $\lambda_2$:

\[
\begin{align*}
\lambda_2'(\text{Key})(X,1) & : \text{ Key on X.} \\
\lambda_2'(\text{Key})(X,0) & : \text{ not Key on X.}
\end{align*}
\]

4.2.5 Equality of relations

Notice that questions such as "are two relations equal?" become impossible to express with this translation of set abstractions. If we attempted to pose a query such as:

\[
\text{..., \{ (X, Y) | Y=X+X \} = \{ (U, V) | V=U*2 \},...}
\]

then with our mapping into $\lambda$-expressions, this condition is actually interpreted as something equivalent to:

\[
\text{..., '$\lambda_x'='\lambda_y',...}
\]

which is false even though the two relations are actually equal. To show that these relations were equal we would require higher order unification between the two set abstractions which we prefer to avoid. In other words we are restricted to manipulating the names or identifiers of relations if we wish to preserve a first order semantics for our language. However this is sufficient for us to be able to use those relations.

The representations of set abstractions and $\lambda$-expressions by parameterized compound names is highly reminiscent of the implementation of $\lambda$-expressions as function closures in functional programming systems. The arguments of the compound name form the environment part of the closure and the function symbol identifies which code fragment represents the $\lambda$-body.

Our representation of function closures actually optimizes on the standard implementation in that the environment parts of our closures are flat (only one level
of arguments is needed in the $\lambda$ name) whereas in order to cope with multiple lexical levels or expressions environments are typically tree structured in functional systems. The flat representation improves performance since the access to the values of free variables in a $\lambda$ body is more direct and garbage collection problems are avoided; however it may be slightly more expensive to construct our form of closure.
Chapter 5: Semantics of class template programs

§ 5.0 Introduction

In Chapter 2 we saw the power of class template programming and gave it an informal justification. In this chapter we are now concerned with the semantics of this formalism. We formally state the syntax of class template programs. A special proof theory shows how we can derive the answers to queries. The proof theoretic semantics is justified by virtue of a mapping from class template programs into standard first order clauses. We will show the soundness and completeness of our inference procedure in terms of standard resolution. The overall aim is to verify that we can inherit properties of standard logic programming in our class template extension.

5.0.1 The Fundamental intuition

Conventional object oriented programming cannot be said to have a particularly strong mathematical foundation. This lack of rigour represents both a problem and an opportunity for us in that the target which we are trying to reach from logic programming is ill-defined and yet therefore we have some freedom in how to integrate logic programming and object oriented programming.

This is in marked contrast with functional programming which we covered in Chapter 4 where the mathematics of equations is much better founded and when combining logic and functional programming the primary problem is integrating this mathematics with the mathematics of logic programming.

In class template notation we have taken a particular view of objects; the fundamental intuition behind our notation is that an object is characterised by what we know to be true of it. Of course while this view may not encompass the whole of object oriented programming (in particular we do not address issues related to assignment) we have shown in earlier chapters some of the expressive power that the notation gives us. Furthermore understanding assignment adequately is a hard problem not confined to the context of object oriented programming.
§5.0 Introduction

A complete class template system may have many objects within it; so whereas a conventional logic program consists of a single global set of true facts, a class template program may have many such sets. These sets may be completely independent, weakly linked (via calls expressed as labelled conditions) or strongly linked (via inheritance). In this sense the class template formalism takes up where standard logic programming leaves off. We are more concerned with whole relations than with individual tuples of relations.

This difference in scale is a key reason why class template notation is so useful. Our object oriented programming view is complementary with conventional logic programming, it is not an alternative. Moreover simply because we have a change of scale does not mean we wish to abandon logic, perhaps the main result of this chapter is that class template programs are still first order logic programs.

5.0.2 The approach to understanding

After having established the set of allowable formulae in a class template program we introduce a number of inference rules. These rules are based on the resolution principle for standard clauses although they are modified in view of the different syntax.

It is important to be able to establish the soundness and completeness of these inference rules. We do this by constructing a mapping from class template programs to a subset of standard clausal form and showing that the mapping is preserved by resolution.

After having established the main result we also show that a set of standard clauses is isomorphic to the set set when it is embedded in a class template program. This property distinguishes our class template formalism from other extensions to logic programming which may only have a 'purely translational' semantics. As an example of such a language we saw in the introduction that BASIC programs can be translated into clauses. However, we cannot embed clauses in a BASIC program in such a way that when translated they are isomorphic to the original clauses.
5.0.3 Class template programs

A class template program consists of a set of class templates, each of which consists of a class body and/or a set of class rules. A class body is a set of axioms enclosed by '{'}'s and prefixed by an identifying label term:

label: {
    axiom\_1.
    axiom\_2.
    ...
    axiom\_n.
}

The label term may be a constant or a complex term. In the case of a complex term variables in the label may be shared with the axioms enclosed in the body. Any variables occurring in the label are universally quantified across all the axioms in the class body. Other variables are universally quantified across the individual axioms.

Each axiom in the class body is of the form:

\[ H: \neg B. \]

where \( H \) is a predication and \( B \) is a condition. In the case that the body is empty then we can use the shorter form:

\[ H. \]

A condition is one of:

i) a predication of the form \( p(t_1, \ldots, t_n) \),

ii) a conjunction of conditions of the form \( C_1, C_2, \ldots, C_m \)

iii) a disjunction of conditions of the form \( C_1; C_2; \ldots; C_m \)

iv) a negated condition of the form \( \neg C \)

v) a labelled condition of the form \( L: C. \)

Variables in a class template program are denoted by Uppercase symbols or symbols whose first character is "_".
§5.1 A proof theory for class template programs

Class rules are used to describe inheritance between class template programs. There are two types of class rule: a normal rule written as:

\[ f(l_1, ..., l_t) \leq m(m_1, ..., m_m) \]

or the overriding class rule which is written as:

\[ f(l_1, ..., l_t) < m(m_1, ..., m_m) \]

This second form of the class rule includes an overriding or default interpretation: relations defined locally in a class body override any relations with the same name which might be inherited with the class rule.

Finally, there are two special keywords in our notation: self and super. The self keyword always refers to the original label (prior to the application of any class rules) associated with any condition, and super refers to those definitions which result from inheritance rather than the complete class template.

§5.1 A proof theory for class template programs

Given a class template program and a query over that program we wish to be able to determine instances of the query which are logical consequences of the program — or completed program in the case of a program with negated conditions. In order to do that we introduce a number of inference rules which allow us to derive new queries from old ones.

Since a class template program consists of a collection of labelled programs rather than a set of clauses in conventional logic programming any top-level query of a class template program must actually take the form of a labelled condition:

\[ L: C? \]

Where \( C \) is a condition. Queries and sub-goals which arise as part of solving such a query may not directly have a label; however as we shall see we have rules which allow us to give a label to any unlabelled condition.

In order for us to be able to handle the self keyword we actually assign two
labels to every condition: the explicit or actual label and the self label which is usually implicitly identified. We write the complete labelled condition by subscripting the actual label with the self label. So a query to a class template program may be represented as:

\[ L_S : Q \]

where \( L \) is the actual label, \( S \) is the self label and \( Q \) is a condition. Where a query is a top-level query then the self label is the same as the actual label:

\[ L_L : Q \]

We will also use the notation

\[ H: \neg B \in L \]

to denote that the axiom \( H: \neg B \) is in the class body associated with label \( L \).

An initial class query is a labelled condition. Notice that it is important for the condition to be labelled since in a class template all the individual axioms are associated with a label, and if we do not have a label for our initial query then we cannot identify which class template to invoke.

A class sequent is a sequence of conditions starting with an initial class query such that each condition (apart from the initial class query) is derived from the previous condition using one of the inference rules outlined above.

A class proof is a class sequent which is terminated by the empty query.

5.1.1 The distribution of labels

Our first group of inference rules relate to the distribution of labels in a condition. Given a labelled condition of the form \( L.C \) we can redistribute the label \( L \) across the connectives that appear in the condition \( C \). Ultimately this may mean that the only labelled conditions within the query are labelled predcations.

We actually have five rules which allow us to distribute labels in a query:
§5.1 A proof theory for class template programs

\[ I_i \quad \mathcal{L}_S : (A, B) \Rightarrow \mathcal{L}_S : A, \mathcal{L}_S : B \]

\[ I_{ii} \quad \mathcal{L}_S : (A; B) \Rightarrow \mathcal{L}_S : A; \mathcal{L}_S : B \]

\[ I_{iii} \quad \mathcal{L}_S : (\neg A) \Rightarrow \neg \mathcal{L}_S : A \]

\[ I_{iv} \quad \mathcal{L}_S : (\text{super} : A) \Rightarrow \text{super} (L) \mathcal{L}_S : A \]

\[ I_v \quad \mathcal{L}_S : (M : A) \Rightarrow \mathcal{M} \mathcal{M} : A \]

This last rule captures the notion that explicitly labelled conditions retain that explicit label and that it is only in unlabelled conditions that predications are assigned labels via an inference rule. It also shows how the self label is identified from an actual label.

The rule for distributing a label across the \text{super} label leaves a special 'marker' label which is interpreted by a variation of the class rule inference rule that we shall see below.

Given these rules we can rewrite any labelled query into an equivalent one where the only labelled formulae are predications and moreover each predication has one label. For example, given the labelled condition

\[ \mathcal{L}_S : (A, \neg (M : B; C)) \]

where \(A, B\) and \(C\) are predications, we can rewrite it thus:

\[ \mathcal{L}_S : A, \mathcal{L}_S : \neg (M : B; C) \quad \text{by } I_i \]

\[ \mathcal{L}_S : A, \neg \mathcal{L}_S : (M : B; C) \quad \text{by } I_{iii} \]

\[ \mathcal{L}_S : A, \neg (\mathcal{L}_S : (M : B); \mathcal{L}_S : C) \quad \text{by } I_{ii} \]

\[ \mathcal{L}_S : A, \neg (\mathcal{M} \mathcal{M} : B; \mathcal{L}_S : C) \quad \text{by } I_v \]

5.1.2 Class body inference rule

Given a condition in the form of a labelled predication we can reduce it in one of two ways: by replacing the label or by replacing the predication. The class body inference rule shows how we can use an axiom from a class body to replace the predication by a simpler condition. The class rule inference rule (see below) shows how we can replace the label.

The class body inference rule is a variation on the standard resolution principle [Robinson'65]. We have to extend it slightly to allow for unifying labels and to
incorporate self.

\[ I_{vi} \quad \mathcal{H} : - C \in L \land Q_1, \ldots, M_S : P, \ldots, Q_n \Rightarrow (Q_1, \ldots, M_S : \text{self}/S, \ldots, Q_n) \theta \]

provided that \( \theta \) is a m.g.u. and that \( (L : \mathcal{H}[\text{self}/S]) \equiv (M : P) \theta \), where \( \mathcal{E}[X/T] \) means the expression \( \mathcal{E} \) with each occurrence of \( X \) replaced by \( T \), in particular \( C[\text{self}/S] \) means replace all occurrences of the self keyword by the self label \( S \). In effect there is an 'extra' substitution of the form \( \{\text{self}/S\} \) at the beginning of the unification (and of course which is still there at the end) between the head of the axiom and the predication.

A simple variation of this inference rule allows us to reduce conditions using axioms which have no body:

\[ I_{vi_h} \quad \mathcal{H} : L \land Q_1, \ldots, Q_{i-1}, M_S : P, Q_{i+1}, \ldots, Q_n \Rightarrow (Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n) \theta \]

provided that \( (L : \mathcal{H}[\text{self}/S]) \equiv (M : P) \theta \) where \( \mathcal{H} \) is an assertion in the class body for \( L \).

### 5.1.3 Class rule inference rules

The class rule inference rules state how we might use the various kinds of class rule in a class template program to replace the label in a labelled predication. Recall that there are two types of class rule: the normal class rule and the overriding class rule. These give rise to two class rule inference rules.

The normal class rule is the simpler to describe:

\[ I_{vii} \quad L \subseteq \mathcal{K} \land Q_1, \ldots, M_S : P, \ldots, Q_n \Rightarrow (Q_1, \ldots, M_S : P, \ldots, Q_n) \theta \]

provided that \( L \equiv \mathcal{K} \theta \) where \( \theta \) is a m.g.u. and \( P \) is a predication. Notice that the self label is not changed by using this inference rule. In fact one may have to use several of these rules before being able to reduce the predication using a class body inference step; and we must be able to recover the original label with self.

With the overriding class rule we have to add an extra constraint on the predicate symbol of the predication to make sure that any locally defined programs are not inherited:
§5.1 A proof theory for class template programs

$I_{viii} \quad L \leftarrow \mathcal{K} \land Q_1, \ldots, M_5; t_{1, \ldots, t_m}, \ldots, Q_n \Rightarrow (Q_{i_1}, \ldots, K_5; P(t_{1, \ldots, t_m}, \ldots, Q_n) \theta$

provided that $L \theta = M \theta$ where $\theta$ is a m.g.u. and that there does not exist an axiom of the form $T(a_1, \ldots, a_m); -c \in L$.

There are two further variations of the class rule inference rule which handle the super label. Recall that when a condition is labelled with super then an inheritance step must be applied to reduce the condition. We therefore have the following rules for reducing super which complement the label distribution rule $I_{iv}$ that we saw earlier:

$I_{ix} \quad L \leftarrow \mathcal{K} \land Q_1, \ldots, \text{super}(M), S: P_t, \ldots, Q_n \Rightarrow (Q_{i_1}, \ldots, K_5; P_t, \ldots, Q_n) \theta$

provided that $L \theta = M \theta$ where $\theta$ is a m.g.u. and $P$ is a predication.

$I_{ix} \quad L \leftarrow \mathcal{K} \land Q_1, \ldots, \text{super}(M) S: t_{1, \ldots, t_m}, \ldots, Q_n \Rightarrow (Q_{i_1}, \ldots, K_5; t_{1, \ldots, t_m}, \ldots, Q_n) \theta$

provided that $L \theta = M \theta$ where $\theta$ is a m.g.u. and that there does not exist an axiom of the form $T(a_1, \ldots, a_m); -c \in L$.

5.1.4 The logical connective inference rules

We allow in our queries and axioms conditions which may be complex, including disjunction and negations. We must therefore also have inference rules to allow us to solve queries containing these complex conditions. However these are essentially standard inference rules recast in terms of labelled conditions; we shall not dwell on them unduly.

$I_{x_ia} \quad Q_1, \ldots, (A; B), \ldots, Q_n \Rightarrow Q_1, \ldots, A, \ldots, Q_n$

$I_{x_ib} \quad Q_1, \ldots, (A; B), \ldots, Q_n \Rightarrow Q_1, \ldots, B, \ldots, Q_n$

$I_{x_ii} \quad Q_1, \ldots, (A, B), \ldots, Q_n \Rightarrow Q_1, \ldots, A, B, \ldots, Q_n$

As for negated conditions we shall assume (though not necessarily rely on) negation-by-failure.

$I_{x_iii} \quad Q_1, \ldots, Q_{i-1}, \neg Q_i, Q_{i+1}, \ldots, Q_n \Rightarrow \text{false}$ if $C? \text{succeeds else}$

$Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n$
if all possible proofs of \( C? \) reduce to \textit{false}, assuming that \( Q \) is ground.

One of our aims in this chapter is to show that the inference rules are sound. We shall do so by constructing a mapping from class template programs into standard first order clauses and showing that inference in class proofs is reflected by resolution steps in the mapped clauses. Since we know that resolution is sound, we can rely on this property to show that our inference rules are sound also. Clearly the mapping that we choose is of some importance.

\section*{§5.2 Mapping class template programs into clausal form}

As we map class template programs into conventional clauses we simultaneously provide a semantics of our programs: the meaning of a class template program is defined by the translated clauses. However our main purpose in introducing this mapping is to justify the inference rules that we have developed above.

We shall look at a mapping where we map each axiom inside a class body into a separate first order clause, and we similarly map class rules into clauses. In the second mapping we give a reading in terms of higher order functions and relations.

The reader will recall that formulae such as

\[ L_S: P \]

where \( L \) is the actual label, \( S \) is the self label and \( P \) is a predication or condition are central in our inference procedures. The fact that there are three components suggests that a reasonable characterization of a labelled condition in terms of classical clausal form is as a triple, with each element being represented by a term. So we might, for example, represent the labelled predication:

\[ \text{person}(30)_{\text{tom}}:\text{likes}(\text{john}) \]

as the class triple\textsuperscript{1}:

\[ \text{class(person}(30), \text{tom, likes}(\text{john})) \]

In general, given a labelled condition of the form \( L_S: C \) we represent it as the triple

\footnote{The choice of the \texttt{class} predicate symbol here is purely arbitrary.}
§5.2 Mapping Class template programs into clausal form

In this representation we are using terms to denote labels (both actual and self labels) and we are also using terms to denote conditions and predications. We shall also use the special function symbols "", "", "-", and "¬" within \( C \) to denote conjunctions, disjunctions and negations respectively. For example, a class query such as

\[
\text{tom} \bowtie \text{tom}: (\text{likes}(X), X: \text{likes}(\text{logic}))?
\]

is represented by the conventional query over class:

\[
\text{class(tom,tom, (likes(X), X: likes(logic)))}?
\]

Class template programs can be represented by clauses which define the class predicate, with one clause for each axiom in each class body, together with further clauses for class rules and some standard auxiliary clauses as defined below.

Given an axiom in a class body such as:

\[
\text{lab}(l_1, \ldots, l_k) : \{ \ldots \\
\quad P(P_1, \ldots, P_H) : -C_1, \ldots, C_n. \\
\quad \ldots \}.
\]

we construct a clause for class of the form:

\[
\text{class(lab}(l_1, \ldots, l_k), \text{Self}, \text{P}(P_1, \ldots, P_H)) : -
\quad \text{class(lab}(l_1, \ldots, l_k), \text{Self}, (C_1, \ldots, C_n)) .
\]

where \( \text{Self} \) is a new variable not occurring elsewhere in the class body axiom. If the body of the class body axiom is empty then the body of the class clause is also empty:

\[
\text{lab}(l_1, \ldots, l_k) : \{ \ldots \\
\quad P(P_1, \ldots, P_H) . \\
\quad \ldots \}.
\]

becomes:
Chapter 5: Semantics of class template programs

class (\text{lab} (l_1, \ldots, l_k), \text{Self}, p (p_1, \ldots, p_H)) .

The rewrite to \text{class} clauses involves bringing the quantified variables associated with the class label to the individual axioms inside the class body. Those variables which were quantified by the whole class body are now simply quantified within the individual \text{class} clause. We can do this because of the equivalence:

$$\forall x, y (\forall u, v : \Psi_1) \land \cdots \land (\forall u, v : \Psi_n) \equiv (\forall u, v, x, y : \Psi_1) \land \cdots \land (\forall u, v, x, y : \Psi_n)$$

where \(\Psi_i\) are sentences in which \(x, y, u, v\) are free variables.

If the class body axiom contained any occurrences of the \text{self} keyword then these are represented in the \text{class} clause as further occurrences of the \text{Self} variable. So, for example if the class body axiom was of the form:

\[
\text{lab} (l_1, \ldots, l_k) : \{ \\
\quad p (p_1, \ldots, p_H) : \neg C_1, \ldots, \text{self}: C_1, \ldots, C_n . \\
\quad \ldots \} .
\]

or if a given condition mentioned the \text{self} keyword as a term as in:

\[
\text{lab} (l_1, \ldots, l_k) : \{ \\
\quad p (p_1, \ldots, p_H) : \neg C_1, \ldots, p_j (t_{j_1}, \ldots, \text{self}, \ldots, t_{j_n}), \ldots, C_n . \\
\quad \ldots \} .
\]

then the corresponding \text{class} clauses would be

\[
\text{class(} \text{lab} (l_1, \ldots, l_k), \text{Self}, p (p_1, \ldots, p_H) \text{)} :- \\
\text{class(} \text{lab} (l_1, \ldots, l_k), \text{Self}, (C_1, \ldots, \text{Self}: C_1, \ldots, C_n) \text{)} .
\]

and

\[
\text{class(} \text{lab} (l_1, \ldots, l_k), \text{Self}, p (p_1, \ldots, p_H) \text{)} :- \\
\text{class(} \text{lab} (l_1, \ldots, l_k), \text{Self}, \\
\quad (C_1, \ldots, p_j (t_{j_1}, \ldots, \text{Self}, \ldots, t_{j_n}), \ldots, C_n) \text{)} .
\]

respectively.

A class rule is represented by a pair of clauses for \text{class}. In the case of a normal
§5.2 Mapping Class template programs into clausal form

class rule such as:

\[
\text{lab}(l_1, ..., l_k) \leq \text{mab}(m_1, ..., m_n)
\]

we would get the pair of clauses:

\[
\text{class}(\text{lab}(l_1, ..., l_k), \text{Self}, \text{Atom}) : \neg\\
\text{class}(\text{super}(\text{lab}(l_1, ..., l_k)), \text{Self}, \text{Atom}) . \quad (1)
\]
\[
\text{class}(\text{super}(\text{lab}(l_1, ..., l_k)), \text{Self}, \text{Atom}) : \neg\\
\text{class}(\text{mab}(m_1, ..., m_n), \text{Self}, \text{Atom}) .
\]

where \text{Self} and \text{Atom} are new variables not occurring elsewhere in the original class rule. We only need one copy of \text{class} clause (1) even if there are several class rules, although the extra clauses will do no harm they are redundant. We shall see later that we need to split a class rule into two \text{class} clauses in order to correctly implement super inheritance.

The translation of the overriding form of class rule is a little more complex since we must incorporate the tests required to filter out the locally defined relations. So if a label had the ‘local predicates’ \text{l}_1, \text{l}_2, ..., \text{l}_k of arities \text{a}_1, \text{a}_2, ..., \text{a}_k respectively then an overriding class rule for the label of the form:

\[
\text{lab}(l_1, ..., l_k) \leq \text{mab}(m_1, ..., m_n)
\]

would be mapped to the \text{class} clause:

\[
\text{class}(\text{super}(\text{lab}(l_1, ..., l_k)), \text{Self}, \text{Atom}) : \neg\\
\text{class}(\text{mab}(m_1, ..., m_n), \text{Self}, \text{Atom}),\\
\text{functor}(\text{Atom}, \text{P}, \text{A}),\\
\neg \text{member}(\text{P}/\text{A}, \{l_1/\text{a}_1, l_2/\text{a}_2, ..., l_k/\text{a}_k\}).
\]

(Recall that there will also be at least one clause for \text{class} equivalent to clause (1) above). \text{functor} is a predicate which is found in many Prolog systems. It behaves as though it were defined by a set of assertions:

\[
\text{functor}(\text{likes}(_), \text{likes}, 1).
\]
\[
\text{functor}(\text{likes}(_,_), \text{likes}, 2).
\]
\[
... 
\]
\[
\text{functor}(\text{constant}, \text{constant}, 0).
\]
\[
... 
\]
There is a tuple in the functor relation for each constant symbol, function symbol and predicate symbol in the program. functor is commonly regarded as a 'meta-logical' predicate although in our view it easily justified in purely first order terms via a finite axiom schema.

The member predicate is standard list membership for example as defined by the clauses:

\[
\text{member}(X, [X|\_]). \\
\text{member}(X, [\_|L]): \neg \text{member}(X, L).
\]

The filter condition

\[ \neg \text{member}(P/A, [l_1/a_1, l_2/a_2, ... , l_x/a_x]) \]

is true if the term \(P/A\) (i.e. the relation being inherited) is not on the list \([l_1/a_1, l_2/a_2, ... , l_x/a_x]\) (i.e. not a locally identified predicate). Clearly we are relying on negation for the definition of overriding inheritance.

Given the intention to use the class template programs as a programming language, we are likely to prefer the use of negation-by-failure as our negation. Moreover we are guaranteed the safety of this negated condition since the variables in the member condition are all governed by the functor condition (under Clark's completion semantics [Clark'78] a negated condition must be ground at the time of the negative proof for negation-by-failure to be equivalent to logical negation).

In order to support the various inference rules for label distribution etc. we also need some additional clauses in the class program. These clauses must be added to the clauses derived from the class template program itself.

\[
\begin{align*}
C_i & \quad \text{class}(L, S, (A, B)) \leftarrow \text{class}(L, S, A), \text{class}(L, S, B). \\
C_{ii} & \quad \text{class}(L, S, (A; B)) \leftarrow \text{class}(L, S, A). \\
C_{iii} & \quad \text{class}(L, S, (A; B)) \leftarrow \text{class}(L, S, B). \\
C_{iv} & \quad \text{class}(L, S, \neg(A)) \leftarrow \neg \text{class}(L, S, A). \\
C_v & \quad \text{class}(L, S, \text{super}: A) \leftarrow \text{class}(\text{super}(L), S, A). \\
C_{vi} & \quad \text{class}(L, S, M:A) \leftarrow \text{class}(M, M, A).
\end{align*}
\]
§5.2 Mapping Class template programs into clausal form

5.2.1 Equations and class templates

Since we have argued that equations and expressions are useful, we would like to be able to combine the equational formalism with our class templates. We saw in Chapter 4 how we can translate equations and clauses containing expressions into clauses which do not, and which still 'compute' the same answers.

If we have an equation within a class body then we can simply combine the earlier translation with the translation for class templates. The first will result in a clause within the class body, and the second will translate this into a normal class clause. So, for example, if we had the class body:

\[(X,Y):\{\]
  \[\begin{align*}
  x\_\text{coord}&=X. \\
  y\_\text{coord}&=Y. \\
  \ldots
  \end{align*}\]
\[
\}
\]

then the equations for \(x\_\text{coord}\) and \(y\_\text{coord}\) would first be translated to:

\[(X,Y):\{\]
  \[\begin{align*}
  x\_\text{coord}^*(X). \\
  y\_\text{coord}^*(Y). \\
  \ldots
  \end{align*}\]
\[
\}
\]

and then, by applying our class template translation, we obtain the class clauses:

\[
\text{class}((X,Y), \text{Self}, x\_\text{coord}^*(X)).
\]
\[
\text{class}((X,Y), \text{Self}, y\_\text{coord}^*(Y)).
\]

An expression may make a reference to a function defined in another class template; recall that we extended the expression syntax to allow a labelled expression which had this rôle:

\[
\text{lab}(l_1, \ldots, l_k): \text{fun}(e_1, \ldots, e_n)
\]

By a simple extension of the definitional sentence for functions we can construct an equivalent for labelled expressions:

\[
\text{lab}(l_1, \ldots, l_k): f(e_1, \ldots, e_n) = X \Leftrightarrow \text{lab}(l_1, \ldots, l_k): f^*(e_1, \ldots, e_n, X)
\]
This allows us to translate conditions containing labelled expressions into equivalent conjunctions which have a labelled condition to control the value of the variable.

Higher order constructs such as $\lambda$-expressions and set abstractions can be similarly handled.

§5.3 The soundness of class template inference

Our strategy in showing the soundness of our inference procedure is to demonstrate that each inference rule when applied to a labelled query is paralleled by normal resolution steps in the mapped set of class clauses. Since we know that resolution itself is sound then if we can show that our inference can be 'reduced' to resolution then it too is sound.

In other words given a class sequent $Q_1, \ldots, Q_n$ then there is a corresponding conventional sequent $M(Q_1), \ldots, M(Q_n)$ where each $M(Q_j)$ is the result of mapping the corresponding $Q_j$. Furthermore if there is a class proof then there must also be a conventional proof in the mapped clauses.

To show this we proceed by induction on the length of the sequent. We take for the base case the original query $Q_0$ and the mapping of this initial query into a standard query for class: $M(Q_0)$.

Now, suppose that we have a non-empty class sequent $Q_1, \ldots, Q_i$ and corresponding conventional sequent $M(Q_1), \ldots, M(Q_j)$ then suppose we wish to extend the class sequent with a new labelled query $Q_{i+1}$. Then by definition there must be an inference rule $I$ in $I$ — $I_{x_i}$ which we wish to apply to some $Q_k$. We must show that $M(Q_1), \ldots, M(Q_j), M(I_k(Q_k))$ is also a sequent. We do that by showing that $M(I_k(Q_k))$ is a logical consequence of $M(Q_1), \ldots, M(Q_j)$, i.e. that the inference step that we applied was valid; which it will be if we can reduce the class inference to conventional resolution.

In general for each of our inference rules $I$ we wish to establish the equality:

$$M(I(Q)) = R^*(M(Q))$$
§5.3 The soundness of class template inference

where $\mathcal{R}^*$ represents the application of one or more steps of standard resolution and $\mathcal{M}$ is the mapping function defined above.

5.3.1 Soundness of label distribution

The label distribution rules take the form:

$$\mathcal{L}_S : (A \oplus B) \Rightarrow \Phi(\mathcal{L}_S : A, \mathcal{L}_S : B)$$

where $\oplus$ is one of ",", ",", "\&" or "\&" and $\Phi$ is some function (equal to $\oplus$ in the cases of ",", ",," and "\&"). These inference rules are reflected by corresponding class clauses. For example inference rule $I_i$ is

$$I_i \quad \mathcal{L}_S : (A, B) \Rightarrow \mathcal{L}_S : A, \mathcal{L}_S : B$$

If we were to map a query of the form $\mathcal{L}_S : (A, B)$ into standard form we would obtain the class query:

$$\text{class}(L, S, (A, B)) ?$$

in other words $\mathcal{M}(\mathcal{L}_S : (A, B)) = \text{class}(L, S, (A, B))$ for all labels $L, S$ and conditions $A$ and $B$. If we now use the class clause $C_i$ which matches this query we obtain a new query consisting of a pair of class conditions:

$$\text{class}(L, S, A), \text{class}(L, S, B) ?$$

However if we apply $\mathcal{M}$ to the right hand side of $I_i$ we obtain the same pair. I.e. we have shown

$$\mathcal{M}(I_i[\mathcal{L}_S : (A, B)]) = \mathcal{M}(\mathcal{L}_S : A), \mathcal{M}(\mathcal{L}_S : B)$$

Similar reasoning applies to each of the label distribution rules $I_i$ through $I_{ip}$. Of course we should not be too surprised at this since this is how we set up the extra class clauses.
5.3.2 The soundness of the class body inference rules

Recall that there are two inference rules which relate to inference within a class body. The main one takes the form:

\[ I_{vi} : H: \neg C \in L \land Q_1 \ldots M_S: T \ldots Q_n \Rightarrow (Q_1 \ldots L_S: C/\text{Self}/S) \ldots Q_n \theta \]

where \((L: H[\text{Self}/S]) \equiv (M: T, \theta)\). In order to justify this rule we shall apply the mapping function to the inputs to the rule, perform the resolution step and show that this is the same (possibly up to the renaming of variables) as the mapped result of the rule.

The class clause which results from mapping the class body axiom is:

\[
\text{class}(L, \text{Self}, H) : - \text{class}(L, \text{Self}, C) .
\]  

(2)

and the mapped form of the class query is:

\[
\text{class}(L_1, S_1, C_1) , \ldots, \text{class}(M, S, T) , \ldots, \text{class}(L_n, S_n, C_n)
\]

(3)

respectively, where each \(Q_i\) is of the form \(L_i S_i^\top\). Suppose now that the condition

\[
\text{class}(M, S, T)
\]

unifies with the head of (2) with m.g.u. \(\theta'\). If we perform the resolution with this m.g.u. then we obtain the new class query:

\[
(\text{class}(L_1, S_1, C_1) , \ldots, \text{class}(L, S, C) , \ldots, \text{class}(L_n, S_n, C_n)) \theta'
\]

(4)

Recall that under the mapping the \text{Self} keyword is represented by the \text{Self} variable in clause (2). As part of this resolution step therefore all occurrences of the \text{Self} variable in (2) would have been replaced by \(S\). Thus in (4) the new condition \text{class}(L, S, C) has had all occurrences of \text{Self} replaced by \(S\) also.

We could equally have rewritten the \(\theta'\) m.g.u. as a composition of \(\text{Self}/S\) and the remainder which we shall call \(\theta\). I.e. we have the equality:

\[
\theta' = \theta \cup (\text{Self}/S)
\]
Since \texttt{Self} is a variable which originally only occurred (after renaming of variables) in clause (2) and does not occur in the query (3) we can rewrite the resolvent (4) without loss as:

\[(\text{class}(L_1,S_1,C_1),\ldots,\text{class}(L,S,C)\langle\text{Self}/S\rangle,\ldots,\text{class}(L_n,S_n,C_n))\theta\]

which is exactly the same as the result of mapping the resultant of our class body inference rule \texttt{I_{bi}} into a class query.

This shows that the class body inference rule can be justified by a resolution step in standard clausal form. If the inference rule were not sound then it would be possible to perform a class body inference step which forced an incorrect resolution step over the class clauses.

The second variety of class body rule is simply a special case of the main one where the body of the class body axiom is empty. The proof of its soundness flows easily from the main proof.

### 5.3.3 The soundness of class rule inference

There are four inference rules which are used to describe the process of inheritance. These relate to the 'normal' inheritance rule, the 'overriding' inheritance rule and the two 'super' variations of these. As with the class body inference rules we shall appeal to the mapping into class clauses in order to justify these inference rules.

Recall that the standard inheritance rule is:

\[I_{vii} \quad L \Rightarrow K \land Q_1 \ldots M_S : P_1 \ldots P_n \Rightarrow Q_1 \ldots K_S : P_1 \ldots P_n \theta\]

provided that \(L\theta = M\theta\). If we apply our mapping to the class rule we obtain the set of clauses for class:

\[
\begin{align*}
\text{class}(L,\text{Self},\text{Atom}) : & - \text{class}(\text{super}(L),\text{Self},\text{Atom}). \quad (5) \\
\text{class}(\text{super}(L),\text{Self},\text{Atom}) : & - \text{class}(K,\text{Self},\text{Atom}). \quad (6)
\end{align*}
\]

the class query in the left hand side of \texttt{I_{vii}} is mapped into the class query:
Chapter 5: Semantics of class template programs

class \( L_1, S_1, C_1 \), ..., class \( M, S, P \), ..., class \( L_n, S_n, C_n \) \? \tag{7}

assuming that each \( Q_i \) is of the form \( L_i, C_i \).

If we now resolve clause (5) with the class \( M, S, P \) condition in (7) we obtain the new query (assuming of course that they unify):

\[
\text{class}(L_1, S_1, C_1), \ldots, \text{class}(\text{super}(L), S, P), \ldots, \text{class}(L_n, S_n, C_n) \theta \tag{8}
\]

This resolution step can only be performed if the predications class \( M, S, P \) and class \( L, \text{Self}, \text{Atom} \) unify (with m.g.u. \( \theta \)). In practice this means that the terms \( L \) and \( M \) must unify. This is precisely the condition that the labels \( L \) and \( M \) unify prior to the application of the class rule inference rule (furthermore the m.g.u.'s will be the same except for the additional substitutions \( \text{Self}/S \) and \( \text{Atom}/P \)).

In order to achieve the same effect as the class rule inference rule we must also resolve the class \( \text{super}(L), S, P \) condition in query (8) with the class clause (6). We can guarantee that we will be able to perform this resolution because of the way that we constructed (6). Furthermore we can say that there will be no further substitutions for variables in the label term \( L \). The resolvent from this second resolution step is the new class query:

\[
\text{class}(L_1, S_1, C_1), \ldots, \text{class}(\mathcal{K}, S, P), \ldots, \text{class}(L_n, S_n, C_n) \theta \tag{9}
\]

which is also what would be obtained if we applied \( \theta \) to the labelled query \( Q_1, \ldots, Q_n \) and mapped the result into a class query. This shows that the standard class rule inference rule is also sound.

The overriding class rule is similar to the normal class rule (and hence the proof of its soundness), with the extra requirement of filtering out the locally defined predicates. Recall that the translation of an overriding class rule such as

\[ L<< \mathcal{K} \]

is

\[
\text{class}(L, \text{Self}, \text{Atom}) :- \text{class}(\text{super}(L), \text{Self}, \text{Atom}).
\]
§5.3 The soundness of class template inference

\[
\text{class}(\text{super}(L), \text{Self}, \text{Atom}) \leftarrow \\
\text{functor}(\text{Atom}, P, A), \\
\neg \text{member}(P/A, \{f_1/a_1, \ldots, f_n/a_n\}), \\
\text{class}(K, \text{Self}, \text{Atom}).
\]

(10)

and that the overriding class rule inference rule is:

\[
L \leftarrow \neg (K \wedge Q_1 \ldots M \leftarrow P[a_1, \ldots, a_n]) \Rightarrow (Q_1 \ldots K \leftarrow P[a_1, \ldots, a_n] \ldots Q_n) \Theta
\]

provided that \(L \Theta = M \Theta\) and that there is no clause defining \(P\) of arity \(l\) in the class body for label symbol \(L\). This extra condition is exactly characterised by the conditions in (10) of the form

\[
\text{functor}(\text{Atom}, P, A), \neg \text{member}(P/A, \{f_1/a_1, \ldots, f_n/a_n\})
\]

since, by definition of the mapping into class clauses, the list of pairs \(\{f_1/a_1, \ldots, f_n/a_n\}\) mentions each of the locally defined predicate symbols in the class body for \(L\). I.e. in place of the constraint on the application of the overriding class rule inference we have to perform an extra sub-proof in the class query which implements the constraint.

Notice that this extra sub-proof relies on negation in the class program. However as we noted above we can show that negation-by-failure (which is the most likely form of negation to be actually used) is equivalent to logical negation in this context.

Apart from this extra sub-proof the steps in applying the overriding class rule are identical to the normal inheritance rule. Hence we can also conclude that the overriding class rule is sound.

The final class rule inference rules that we must justify are the rules for interpreting the super label. Recall that we have split the procedure for handling super into two steps: the first step is a label distribution which leaves a 'marker' for super:

\[
I_{iv} \quad L_5: (\text{super}: C) \Rightarrow \text{super}(L_5): C
\]

and the second step involves applying a class rule to this marker. So we obtain the rule:
Chapter 5: Semantics of class template programs

\[ I_{\mathcal{K}} \leq \mathcal{K} \land Q_1, \ldots, \text{super}(\mathcal{M}) \ni \mathcal{P}, \ldots, Q_n \Rightarrow (Q_1, \ldots, \mathcal{K}, \mathcal{P}, \ldots, Q_n)^\theta \]

for the normal class rule, and

\[ I_{\mathcal{K}} \leq \mathcal{K} \land Q_1, \ldots, \text{super}(\mathcal{M}) \ni \mathcal{P}, \ldots, Q_n \Rightarrow (Q_1, \ldots, \mathcal{K}, \mathcal{P}, \ldots, Q_n)^\theta \]

for the overriding class rule. (Assuming that the required preconditions hold of course.)

If we map the labelled query in the left hand side of either of these inference rules into a class query we get:

\[
\text{class}(L_1, S_1, C_1), \ldots, \text{class}(\text{super}(\mathcal{M}), S, \mathcal{P}), \ldots, \text{class}(L_n, S_n, C_n)
\]

by resolving the 'super' condition with the clause (6) (which we obtained as part of mapping the class rules themselves) then we obtain the new class query:

\[
(\text{class}(L_1, S_1, C_1), \ldots, \text{class}(\mathcal{K}, S, \mathcal{P}), \ldots, \text{class}(L_n, S_n, C_n))^\theta
\]

provided that \( L \equiv \mathcal{M} \), and in the case of the overriding class rule providing that the sub-proof for non-membership of the locally defined list of predicates is successful. Clearly this new query is the result of mapping the resultant of the inference rule, and hence this rule too is sound.

Incidentally we can now see why we needed to split the translation of class rules into the two class rules. It makes it easy for us to prevent the application of class body axioms to a super labelled condition since \( \text{super}(L) \) will never unify with \( L \).

§5.4 The completeness of class template inference

By completeness we mean that any justifiable inference is achievable using our class template inference rules. In other words we have sufficiently characterised class template inference so that we do not need any 'extra' rules.

Since we know that resolution is complete we can show completeness by examining the inferences possible with the translated class programs and show that they are achievable with our rules. However we are not necessarily interested
§5.4 The completeness of class template inference

in all possible inferences over the mapped class clauses, merely those which are 'reachable' from class queries. Thus a formal statement of the completeness proposition is:

PropH “Given a class query \( Q_0 \equiv L: Q \) together with a class template program \( P \) and their mapped forms \( M(Q_0) \equiv \text{class}(L, L, Q) \) and \( M(P) \) then if there is a proof of \( M(Q_0) \) using \( M(P) \) and resolution then there is a class proof of \( Q_0 \) using \( P \) and the class template inference rules.”

We shall prove this by induction on the length of the proof. Trivially, if the initial class query is of the form \( L: \text{true} \) then we are already done. Otherwise consider the partial proof consisting of the sequent \( M(Q_0), ..., M(Q_p) \) where (under the induction hypothesis) query \( M(Q_p) \) has a corresponding \( Q_i \) which is 'reachable' from \( Q_0 \) and \( P \). It follows that \( M(Q_j) \) takes the form:

\[
\text{class}(L_0, S_0, C_0), \ldots, \text{class}(L_n, S_n, C_n) ?
\]

since this is the mapped form of any class query. Suppose now that we wish to extend the partial proof to obtain a new query. In order to do so we must resolve one of the class \( (L_j, S_j, C_j) \) \((0 \leq j \leq n)\) conditions with a suitable clause. Given the mapped program \( M(P) \) we can predict the possible input clauses that might match with this query:

There are a number of standard class clauses in all \( M(P) \)s:

\[C_i\] class \((L, S, (A,B)) \) :- class \((L, S, A)\), class \((L, S, B)\).

\[C_{ii}\] class \((L, S, (A,B)) \) :- class \((L, S, A)\).

\[C_{iii}\] class \((L, S, (A,B)) \) :- class \((L, S, B)\).

\[C_{iv}\] class \((L, S, \neg(A)) \) :- \( \neg\)class \((L, S, A)\).

\[C_v\] class \((L, S, \text{super:A}) \) :- class \((\text{super}(L), S, A)\).

\[C_{vi}\] class \((L, S, M:A) \) :- class \((M, M, A)\).

For each axiom in each class body we have a class clause along the lines of:

\[C_{vii}\] class \((\text{lab}(l_1, \ldots, l_k), \text{Self}, p(p_1, \ldots, p_h)) \) :-

\[
\text{class}(\text{lab}(l_1, \ldots, l_k), \text{Self}, (C_1, \ldots, C_n)) .
\]
for each normal class rule we will have the pair of clauses:

\[
C_{viii_a} \quad \text{class}(\text{lab}(l_1, \ldots, l_k), \text{Self}, \text{Atom}) :- \\
\quad \text{class}(\text{super}(\text{lab}(l_1, \ldots, l_k)), \text{Self}, \text{Atom}).
\]

\[
C_{viii_b} \quad \text{class}(\text{super}(\text{lab}(l_1, \ldots, l_k)), \text{Self}, \text{Atom}) :- \\
\quad \text{class}(\text{mab}(m_1, \ldots, m_n), \text{Self}, \text{Atom}).
\]

and for each overriding class rule we will have the pair:

\[
C_{ix_a} \quad \text{class}(\text{lab}(l_1, \ldots, l_k), \text{Self}, \text{Atom}) :- \\
\quad \text{class}(\text{super}(\text{lab}(l_1, \ldots, l_k)), \text{Self}, \text{Atom}).
\]

\[
C_{ix_b} \quad \text{class}(\text{super}(\text{lab}(l_1, \ldots, l_k)), \text{Self}, \text{Atom}) :- \\
\quad \text{class}(\text{mab}(m_1, \ldots, m_m), \text{Self}, \text{Atom}), \\
\quad \text{functor}(\text{Atom}, P, A), \\
\quad \neg \text{member}(P/A, [l_1/a_1, l_2/a_2, \ldots, l_x/a_x]).
\]

If any of the class clauses \(C_i \rightarrow C_{vi}\) matched then it follows that the class \((L_j, S_j, C_j)\) condition must have been one of the following forms:

\[
\text{class}(L_j, S_j, (C_j, D_j)) \\
\text{class}(L_j, S_j, (C_j, \neg D_j)) \\
\text{class}(L_j, S_j, \text{super}:C_j) \\
\text{class}(L_j, S_j, M_j: C_j)
\]

in which case the next query in the proof of \(M(Q_j)\) is one of

\[
\text{class}(L_0, S_0, C_0) \ldots \text{class}(L_j, S_j, C_j) \ldots \text{class}(L_n, S_n, C_n) \\
\text{class}(L_0, S_0, C_0) \ldots \text{class}(L_j, S_j, C_j) \ldots \text{class}(L_n, S_n, C_n) \\
\text{class}(L_0, S_0, C_0) \ldots \text{class}(L_j, S_j, C_j) \ldots \text{class}(L_n, S_n, C_n) \\
\text{class}(L_0, S_0, C_0) \ldots \text{class}(\text{super}(L_j), S_j, C_j) \ldots \text{class}(L_n, S_n, C_n) \\
\text{class}(L_0, S_0, C_0) \ldots \text{class}(M_j, M_j, C_j) \ldots \text{class}(L_n, S_n, C_n)
\]

In each of the above cases there is a corresponding class template inference rule \(I_i \rightarrow I_v\) which would have the same effect. I.e. if one of the clauses \(C_i \rightarrow C_{vi}\) were used to produce the next step in the proof of \(M(Q_j)\) then there is a corresponding inference rule \(I_i \rightarrow I_v\) which induces an analogous step in the class proof of \(Q_0\).
§5.4 The completeness of class template inference

There are three other types of class clauses which might be used to produce the next step in the proof; these arise from class body axioms and the two types of class rule.

If one of the clauses of the form $C_{vii}$ were used to reduce the class $(L_j, S_j, C_j)$ condition then the next query in the proof of $M(Q_0)$ would be of the form:

$$(\text{class}(L_0, S_0, C_0), \ldots, \text{class}(L_j, S_j, (C_j') \ldots, C_n)) \Theta$$

where $\Theta$ is the m.g.u. that results from unifying

$$\text{class}(L_j, S_j, C_j)$$

and

$$\text{class}(\text{lab}(l_1, \ldots, l_k), \text{Self}, p(p_1, \ldots, p_H))$$

which is the same m.g.u. that would result from unifying

$$L_j S_j : C_j$$

and

$$\text{lab}(l_1, \ldots, l_k) \text{Self} : p(p_1, \ldots, p_H)$$

in other words applying a clause from the set of clauses $C_{vii}$ to produce the next query in the proof of $M(Q_0)$ is exactly mirrored by applying a step of inference rule $I_{vii}$ in the proof of $Q_0$.

If one of the clauses $C_{viii_a}$ were to be applied to the class $(L_j, S_j, C_j)$ condition then the next query in the proof of $M(Q_0)$ would be of the form:

$$(\text{class}(L_0, S_0, C_0), \ldots, \text{class}(\text{super}(L_j), S_j, C_j), \ldots, \text{class}(L_n, S_n, C_n)) \Theta$$

where $\Theta$ is the m.g.u. that results from unifying $L_j$ and $\text{lab}(l_1, \ldots, l_k)$ together with the additional substitutions $\{\text{Self}/L_j \land C_j\}$.

In this case we cannot say that the new query is directly the result of mapping a class query, and hence that the resolution step can be mirrored by a class template inference rule. However, we know that in order for the proof of $M(Q_0)$ to successfully terminate the condition class $(\text{super}(L_j), S_j, C_j)$ must eventually
be selected for further reduction. Furthermore we know (by the independence of
the order of selecting sub-goals) that if a proof of $M(Q_0)$ exists then an equivalent
proof exists in which the next step involves the successful resolution of this new
condition with some input clause. In other words we can assume that the next step
in the proof of $M(Q_0)$ involves resolving the $\text{class}(\text{super}(L_j), S_j, C_j)$ with
some input clause.

There are only two possibilities for this: using one of the $C_{viii_6}$ clauses or one of
the $C_{ix_6}$ clauses. If one of the former then, as with the other inference rules, the
resolution step can be mirrored by the application of the class rule inference rule
$I_{viii}$. If one of the $C_{ix_6}$ clauses was used to reduce the condition then the next
class query is of the form:

$$(\text{class}(L_0, S_0, C_0),$$
$$\ldots, \text{class}(\text{mab}(m_1, \ldots, m_m), S_j, C_j),$$
$$\text{functor}(C_j, P, A),$$
$$\neg \text{member}(P/A, \{f_1/a_1, f_2/a_2, \ldots, f_\alpha/a_\alpha\}),$$
$$\ldots, \text{class}(L_n, S_n, C_n)) \theta$$

where $\theta$ is the m.g.u. from the unification.

By a similar argument to that above concerning the reordering of any proof of
$M(Q_0)$ we can assume that the next few steps in the proof involve the solving of the
conditions

$$\text{functor}(C_j, P, A)$$

and

$$\neg \text{member}(P/A, \{f_1/a_1, f_2/a_2, \ldots, f_\alpha/a_\alpha\})$$

and (since we know that we have a proof of $M(Q_0)$ we know that we will be able
to successfully solve these two conditions. The resulting query will look like:

$$(\text{class}(L_0, S_0, C_0), \ldots, \text{class}(\text{mab}(m_1, \ldots, m_m), S_j, C_j), \ldots, \text{class}(L_n, S_n, C_n)$$

which can also be mirrored in the proof of $Q_0$ by a successful application of the
class rule inference rule $I_{viii}$. 
§5.5 Conventional logic programs within class template programs

We stated at the beginning of this chapter that we could map conventional logic programs into class templates without affecting their meaning. It is instructive to see how this can be accomplished in a little more detail. A logic program consists of a set of clauses, each of which is of the form:

\[
\text{pred}_H(t_{H1},...,t_{Hk}) : - \\
\quad \text{pred}_1(t_{l1},...,t_{l_k}), \\
\quad ... \\
\quad \text{pred}_n(t_{n1},...,t_{nk}).
\]

We can take our collection of clauses and put them into the special class body \(\emptyset\):

\[
\emptyset : \{...
\quad \text{pred}_H(t_{H1},...,t_{Hk}) : - \\
\quad \quad \text{pred}_1(t_{l1},...,t_{l_k}), \\
\quad \quad ... \\
\quad \quad \text{pred}_n(t_{n1},...,t_{nk}). \\
\quad ...\}.
\]

A query to this class template program would be represented by the labelled query:

\[
\emptyset : \text{pred}_q(t_{q1},...,t_{qk})?
\]

where \(\text{pred}_q(t_{q1},...,t_{qk})?\) is a query over the original set of clauses.

If we apply our translation to the \(\emptyset\) class template would would get a set of class clauses each of which is of the following form:

\[
\text{class}(\emptyset, \text{Self}, \text{pred}_H(t_{H1},...,t_{Hk})) : - \\
\quad \text{class}(\emptyset, \text{Self}, \text{pred}_1(t_{l1},...,t_{l_k})), \\
\quad ... \\
\quad \text{class}(\emptyset, \text{Self}, \text{pred}_n(t_{n1},...,t_{nk})).
\]

\(\emptyset\) is only special in the sense that it is only going to be used for this purpose.
where \texttt{Self} and \( \emptyset \) do not occur anywhere apart from where we have explicitly indicated. Our labelled query also has a class form:

\[
\text{class}(\emptyset, \emptyset, \text{pred}_q(t_{q_1}, ..., t_{q_n}))?
\]

where the original query was \( \text{pred}_q(t_{q_1}, ..., t_{q_n})? \)

Since the original class template was constructed by enclosing a set of conventional clauses within a class body we know that there are no class rules for \( \emptyset \), nor are there any explicitly labelled conditions.

It should be obvious that these two programs are very similar; they differ in the presence of the \texttt{Self} variable and the \( \emptyset \) label. Each clause in each program has an exactly corresponding clause in the other program (there are no extra clauses in either program). In fact these programs are equivalent in the following sense:

\[
P \models Q \iff C_P \models \emptyset : Q.
\]

where \( P \) is the original logic program, \( Q \) is a query over that program, and \( C_P \) is the class template version of \( P \). We can show this by mapping our class query \( \emptyset : Q? \) into the standard clausal form \( M(\emptyset : Q)? \) and showing that

\[
P \models Q \iff M(C_P) \models M(\emptyset : Q)
\]

which we can do by examining the possible proofs of one query and showing that there are corresponding proofs in the other query.

This result illustrates how it is the case that class template programming is still logic programming when BASIC and DCG's (say) are not strictly logic programming even though they may both have a defensible semantics. In neither BASIC nor DCG's can we embed standard logic programming programs in such a way as to preserve the meaning of the logic sentences.

We also know now that the class template notation is sufficiently expressive that any logic program (and hence any program) can be expressed within the formalism. This in turn means that we can construct special purpose compilers for class template programs directly without necessarily having to translate them first into clauses. This is something we investigate in the next chapter. Of course if we did construct such a compiler system we would have to be faithful to the semantics as
§5.6 Class template programs and higher order functionals

The mapping that we established from class template programs to conventional class clauses is not the only possible mapping. In this section we examine an alternative interpretation based on the use of higher-order functions and set-abstractions introduced in Chapter 1.

We can regard a class template program as a higher order functional from label terms to set abstractions; these representing the programs as defined in the class body. While this scheme involves the use of higher order features such as set abstractions and \( \lambda \)-expressions we saw in Chapter 4 that it is possible, given the constraints on our use of \( \lambda \)'s, to reduce a language containing them into an equivalent first order language.

In Chapter 1 we saw that we can use set abstractions to give us an alternative interpretation of clauses and conditions, in which a clause of the form

\[
p(t_1, ..., t_n) \leftarrow C_1, ..., C_n
\]

can be re-expressed as an equation whose right hand side is a set abstraction:

\[
p = \{ (t_1, ..., t_n) \mid C_1, ..., C_n \}
\]

and a condition of the form

\[
..., g(a_1, ..., a_n), ...
\]

can similarly be expressed as a form of set membership:

\[
..., (a_1, ..., a_n) \in g, ...
\]

or if \( g \) is bound to a set abstraction as:

\[
..., (a_1, ..., a_n) \in \{ (t_1, ..., t_n) \mid C_1, ..., C_n \}, ...
\]

The 'set abstraction' interpretation of class template programs expands on this idea...
by giving an interpretation of a class template as a functional. A function is defined for each label symbol in the complete program, with the various components of a class template appearing in the equations for the label symbol. Any clauses in the class body are represented as arms in a union of set abstractions. So, given a class template such as

```plaintext
person(A) : {  
    likes(X) :- X: age(G), G<10.  
    likes(X) :- X: likes(logic).  
    age(A).  
}
```

we could represent this as the equation:

```plaintext
person(A) = 
\{(likes(X),Self)|X(age(G),Self),G<10\} | |
\{(likes(X),Self)|X(likes(logic),Self)\} | |
\{(age(A),Self)|true\}. 
```

In general, if we have a class body with an associated label

```
lab(l_1, ..., l_k)
```

and an axiom in the class body of the form

```
pred(h_1, ..., h_n) :- C_1, ..., C_m  
```

where each $C_i$ is either of the form

```
\ldots, p(a_1, ..., a_i), \ldots 
```

or

```
\ldots, label: p(a_1, ..., a_i), \ldots 
```

then there is induced in the equation for lab a set abstraction of the form

```
\{(pred(h_1, ..., h_n), Self)| D_1, ..., D_m\}
```

where each $D_i$ is either of the form
§5.6 Class template programs and higher order functionals

..., \text{lab}(l_1, \ldots, l_K) (p(a_1, \ldots, a_l), \text{Self}), ... \\
or \\
..., \text{label}(p(a_1, \ldots, a_l), \text{label}), ... \\

depending on the original form of the corresponding condition \( C_i \). If there is more than one axiom in the class body then the alternatives are expressed through more arms in the union of set abstractions:

\[
\text{lab}(l_1, \ldots, l_K) = \{ (\text{pred}(h_1', \ldots, h_K'), \text{Self}) | D_1, \ldots, D_m \} \ |
\]

\[
... \\
\{ (\text{pred}(h_p', \ldots, h_K'), \text{Self}) | D_p, \ldots, D_m \}
\]

The translation of a class rule depends on its form, a class rule of the form

\[
\text{lab}(l_1, \ldots, l_K) \subseteq \text{mab}(m_1, \ldots, m_0).
\]

simply becomes a new arm in the union:

\[
\text{lab}(l_1, \ldots, l_K) = \ldots \ | \ \text{mab}(m_1, \ldots, m_0) \ldots
\]

The \( \text{mab} \) functional returns a (union) set abstraction which can be applied to the same arguments as \( \text{lab} \) which are the predication tuple and the \( \text{Self} \) argument.

If an overriding class rule is to be interpreted then we get a set abstraction whose body contains inequalities which express the constraint that locally defined programs are not inherited. So, the class rule:

\[
\text{lab}(l_1, \ldots, l_K) \ll \text{lab}(m_1, \ldots, m_0).
\]

is represented by the set abstraction in the equation for \( \text{lab} \):

\[
\text{lab}(l_1, \ldots, l_K) = \ldots \\
| | \{ (\text{Atom}, \text{Self}) \ | \ \text{functor}(\text{Atom}, P, A), \\
\text{member}(P, A, [l_1/a_1, \ldots, l_K/a_K]), \\
\text{mab}(m_1, \ldots, m_0) (A, \text{Self}) \}
\]

where \( [l_1/a_1, \ldots, l_K/a_K] \) is a list of the predicate symbol/arity pairs that are locally defined in the class body for \( \text{lab}(l_1, \ldots, l_K) \).
As with the standard mapping, any occurrences of the self keyword are mapped into further occurrences of the Self parameter.

Using this form of mapping we can regard a class template as a (restricted form of) higher order function from labels to programs. We could also show the soundness and completeness of the class template inference rules using this mapping as the basis. (The proof would be similar, except that some of the class inference rules are effectively applied at compile-time rather than as part of the evaluation itself.) Thus we can establish an interesting relationship between objects and functions!

On the other hand even the casual reader will observe the syntactic complexity of the set abstraction formulation. If we took this reading as our semantics then the class template notation would be fully justified on the grounds of syntactic convenience alone.
Chapter 6: Implementing class templates

It is, and always has been, our intention that the formalisms that we have introduced in earlier chapters should form the basis of a *programming* language. In Chapter 5 we gave a mapping from class template programs to standard clauses. Whilst this mapping is adequate for the purposes of giving a semantics to class templates; it is not a practical means of implementing a viable programming language.

However, we can give a better mapping\(^1\) from class templates into Prolog which would allow class template programs to be executed as efficiently as normal Prolog programs. Such a mapping allows anyone with a Prolog compiler to use our system giving the advantages of class templates to normal Prolog programmers. In this chapter we explore the mapping in some detail. A MacProlog version of the complete preprocessor which implements the mapping is given in Appendix B.

The preprocessor program takes advantage of a number of special features of MacProlog (for example the ability to have variable function symbols and predicate symbols as well as the ability to have complex function symbols). It would therefore take a little effort to remap the preprocessor into standard Prolog. It would also be necessary to implement a new version of the standard Prolog term parser since we haven liberties with that also.

\section*{6.1 A preprocessor for translating class templates into Prolog}

An important consideration for our preprocessor is that class template programs which are really Prolog programs in disguise (i.e. Prolog programs which embedded within class bodies) should execute with comparable efficiency to the same programs ‘outside’ a class template. This means that any performance enhancing features such as indexing and tail recursion optimization which might be available in the host Prolog system should not be lost or absorbed in the translation into Prolog clauses. Since class bodies contain the actual programs used in a query we concentrate on the efficient mapping from class bodies into Prolog clauses.

\footnote{The basis of this mapping was suggested to the author by D.H.D.Warren in a private communication.}
The two main types of indexing that are common in Prolog systems are predicate indexing — where the clauses are sorted by predicate symbol and arity — and argument indexing where the clauses of an individual predicate are indexed by the type and value of some argument (typically the first argument).

The requirement that predicate indexing is preserved by the preprocessor means that individual predicates within class bodies should be mapped to different Prolog predicates. Furthermore, in order to be able to make use of any argument indexing that is available the top-level arguments of an axiom in a class body must also be mapped into the same top-level arguments in the translation. However, in order to implement inheritance, self reference and arguments in labels, we will need to add some extra arguments to each clause in the Prolog code.

Overall, our strategy is to have two types of translated Prolog clauses — the label programs which correspond to the class rules in the class templates and the local programs which correspond to class bodies. Prolog execution of a class template query will switch between the two sets of clauses depending on whether the application program is in 'inheritance mode' or in 'class body mode'.

6.1.1 Class bodies

A class body consists of a set of axioms, each of which is of the form:

\[ \text{lab}(l_1, \ldots, l_k) : \{ \ldots, p(h_1, \ldots, h_n) : - C_1, \ldots, C_m \ldots \} \]

For each such axiom, we generate the Prolog clause:

\[ '\text{lab}:p'(h_1, \ldots, h_n, \text{Lab}, \text{Sf}) : - C_1^{\dagger}, \ldots, C_m^{\dagger}. \]

We shall see shortly how the \( C_i^{\dagger} \) are obtained from the corresponding \( C_i^{\dagger} \). The 'lab:p' predicate symbol of the Prolog target clause is a newly constructed symbol which is a composite of the label symbol lab from the class body and the predicate symbol p from the original axiom. Obviously, when constructing new symbols, it is advisable to ensure that there is no clash between the new symbol and
any pre-existing ones. We do this by inserting the `:` character into the new symbol as well. Whilst this does not guarantee the uniqueness of the generated symbol neither is it likely that that there are any 'normal' predicates which match this form; on the other hand it is useful for system management reasons to have generated names which are related to the original names (for example it helps when listing programs).

The individual conditions $\mathbf{C}_i^+$ in the translated clause are obtained from the conditions in the body of the axiom in a process which is similar to the way that we generated the head. If the $\mathbf{C}_i$ was a normal un-labelled condition:

$$q(t_1, ..., t_j)$$

for which there is a definition of $q$ within the class body then we generate the condition:

$$'\text{lab}:q'(t_1, ..., t_j, Lb, Sf)$$

in the translation. The $Lb$ and $Sf$ variables that we use here (and in the head) are new unique variables that do not occur elsewhere in the axiom; furthermore they are shared with the head. In fact we prefer to use two standard names for these variables: `$_{?lb}?$' and `$_{?sf}?$' respectively. The `$_{?sf}?$' variable corresponds to the value of the self keyword. As a part of the mapping into clauses we also replace any occurrences of the self keyword by extra occurrences of the `$_{?sf}?$' variable.

If the condition of the class body axiom was itself labelled then we choose a different translation, making the label symbol the predicate of the translated condition and the predication an argument of it. So, for example, if the condition were

$$\text{mab}(m_1, ..., m_0) : q(t_1, ..., t_j)$$

then the translated condition is

$$\text{mab}(q(t_1, ..., t_j), \text{mab}(m_1, ..., m_0), \text{mab}(m_1, ..., m_0))$$

We shall see the reason for the double occurrence of the $\text{mab}(m_1, ..., m_0)$ term when we look at the translation of class rules.

We will see how these can be names of variables below.
A translation such as this may not be possible in the case that the label is a variable in the labelled condition. In this case we generate a call of the form:

\[ '?:?'(q(t_1, \ldots, t_j), X, X) \]

(where X is the variable label in question). The '?:?' predicate is described below.

In the case that the explicit label associated with the condition is the \texttt{super} keyword then we rewrite the condition to a special \texttt{super-local} predicate:

\[ 'lab:super'(q(t_1, \ldots, t_j), _?lb?, _?sf?) \]

If a condition is unlabeled, but for which there is no local definition then we translate it as though the call were \texttt{super} labelled.

6.1.2 \textit{Label variables}

The label variables associated with a class body are all collected together into the '_?lb?' variable in each translated Prolog clause. The variable represents the 'actual' label of the condition. It is a term — compound in the case that there are any parameters to the label. Since there may also be occurrences of label variables in the body of a clause we must ensure that these are correctly treated.

If a label variable occurs in a class body axiom then it becomes a variable in the translated clause. However, we also generate an extra condition of the form:

\[ \text{arg}(_?lb?, n, V) \]

where \( n \) is the index of the label variable \( V \) in the label. This \texttt{arg} condition ensures that the clause variable is identified with the appropriate label argument. The \texttt{arg} predicate is a standard Prolog predicate which is usually extremely efficiently implemented in Prolog systems.

We can now see how a complete class body is translated into Prolog clauses:
train(Cn, Sp, Co): {
    speed(Sp).
    colour(Co).
    country(Cn).
    t_of_j(D, T): -
        speed(S),
        T is D/S.
}

would become mapped to the clauses:

'train:speed' (Sp, _?lb?, _?sf?): -arg(_?lb?, 2, Sp).
'train:colour' (Co, _?lb?, _?sf?): -arg(_?lb?, 3, Co).
'train:country' (Cn, _?lb?, _?sf?): -arg(_?lb?, 1, Cn).
'train:t_of_j' (D, T, _?lb?, _?sf?): -
    'train:speed' (S, _?lb?, _?sf?),
    'train:super' (T is D/S, _?lb?, _?sf?).

An alternative to using `arg` conditions to determine label variables is to replace the
' _?lb? ' variable in a clause which has an occurrence of a label variable with a
copy of the label itself. So, the first of our clauses might be translated into

'train:speed'(Sp, train(Cn, Sp, Co), _?sf?).

This translation for label variables is better for assertions than for general rules
since in that case the label term `train(Cn, Sp, Co)` may be repeated many times.
This would, in most Prolog systems, lead to more garbage being generated than in
the `arg` translation. Adding extra `arg` conditions need not affect any tail recursion
optimizations since they always occur before the condition in which the label
variable occurred. Our preprocessor uses the `arg` method consistently for both
assertions and rules.

6.1.3 Labelled calls and class bodies

We observed above that an explicitly labelled condition was translated differently
from an unlabeled condition. This is to allow programs in one class template to call
programs from another class template. Since we have mapped the local programs
within a class body into different local predicates in the Prolog translation we need
§6.1 A preprocessor for translating class templates into Prolog

to allow a non-local call to be mapped to the local definition. We do this by generating a special 'linking' clause for each local predicate in a class body. These linking clauses are of the form:

$$\text{lab}(p(X_1, ..., X_n), _\text{?lb?}, _\text{?sf?}) :=$$

$$'\text{lab:p'}(X_1, ..., X_n, _\text{?lb?}, _\text{?sf?}) .$$

This ensures that the labelled call is mapped into the correct local definition. Of course we rely on Prolog systems' indexing on the first argument to make the selection of the linking clause rapid.

Notice that we do not repeat the actual label of the class body in this clause. The actual (and self) labels are only specified in the original labelled call. This is so that the label term for each labelled call is only ever constructed once — again this is to minimize garbage and to optimize speed.

We observed above that a labelled call in which the label was actually a variable was translated into a call to the '?' program. Just as the label linking clauses link a labelled call to the local definitions, so the '?' program links a variable label condition to the label program. We can define the '??:?:' program as a set of rules each being of the form:

$$'??:?(\text{lab}(X_1, ..., X_k), _\text{?pr?}, _\text{?sf?}) :$$

$$\text{lab}( _\text{?pr?}, \text{lab}(X_1, ..., X_k), _\text{?sf?}) .$$

The '_'?pr?' variable represents the predication of the labelled condition. Since there may be many class templates in a complete program, there will be equally many clauses in the '?' program; by putting the label term as the first argument of the '?' predicate we maximize the potential for indexing.

In a real system which is dynamically changing it may actually be more convenient to implement '?' via the functor primitive:

$$'??:?( _\text{?lb?}, _\text{?pr?}, _\text{?sf?}) :$$

$$\text{functor}( _\text{?lb?}, \text{Lpred}, _),$$

$$\text{Lpred}( _\text{?pr?}, _\text{?lb?}, _\text{?sf?}) .$$

In Prolog systems which do not support the variable predicate symbol the '?' program would need to be defined as:
Chapter 6: Implementing class templates

6.1.4 Class rules

Class rules are 'methods' for solving queries by replacing the label in a condition rather than the predication. A class rule can be used in a number of contexts — a 'new' call to a class template may mention a predicate which is not defined locally within the class body (there might not even be a class body); a local program may fail and through backtracking try any inherited definitions; a class body axiom may mention a condition which is not locally defined and finally a condition may have an explicit super label in which case the local programs are not to be tried anyway.

In order to be able to cater for all these different uses, and in order to be able to correctly and efficiently handle overriding class rules we generate up to two Prolog clauses for each class rule. The first one will be used on an initial entry to the class template — either directly or via an undefined local predicate or super labelled condition.

The second Prolog clause is used in the case that a local program backtracks its way 'out' of the class body. This second group of clauses is constructed from the normal class rules only. The overriding class rules are not used for these clauses since, by definition, such class rules are not applicable for locally defined predicates. This structure, together with a 'cut' in the label linking clauses, allows us to implement inheritance — including overriding inheritance — correctly and efficiently.

A class rule is a rule about the label symbol. When translated into Prolog this means that a class rule becomes a Prolog clause about the label predicate. (We have already seen a simple example of this in the definition of the '??:?' program.) So, a class rule such as:

\[
\text{label}(l_1, \ldots, l_k) \Leftarrow \text{mabel}(m_1, \ldots, m_n)
\]

could be generated into the clause for label:
§6.1 A preprocessor for translating class templates into Prolog

\[
\text{label(}_?\text{pr?},\text{label}(l_1,\ldots,l_k),_?\text{sf?}):-}
\begin{align*}
\text{mabel(}_?\text{pr?},\text{mabel}(m_1,\ldots,m_n),_?\text{sf?})}.
\end{align*}
\]

Where the predication of the labelled query as represented by the '_?pr?' variable is simply passed from the label program to the mabel program. (Or in a declarative style reading: if _?pr? is true of mabel, then it is also true of label). Similarly the '_?sf?' variable passes on to the mabel program the current value of the self keyword.

As we noted above, we actually generate two clauses for each class rule; and for convenience we actually construct them for two local predicates label:super and label:inherit; the second clause will only be generated if the class rule was a normal one:

\[
\begin{align*}
\text{'label:super' (}_?\text{pr?},\text{label}(l_1,\ldots,l_k),_?\text{sf?}):-}
\end{align*}
\begin{align*}
\text{mabel(}_?\text{pr?},\text{mabel}(m_1,\ldots,m_n),_?\text{sf?}).}
\end{align*}
\]

\[
\begin{align*}
\text{'label:inherit' (}_?\text{pr?},\text{label}(l_1,\ldots,l_k),_?\text{sf?}):-}
\end{align*}
\begin{align*}
\text{mabel(}_?\text{pr?},\text{mabel}(m_1,\ldots,m_n),_?\text{sf?}).}
\end{align*}
\]

In order for each local predicate to have access to the inherited definitions we add an extra clause to each local program. This clause simply calls the label:inherit program in the case that the previous clauses for the local predicate fail:

\[
\begin{align*}
\text{'lab:pred' (X_1,\ldots,X_j),_?lb?,_?sf?):-}
\end{align*}
\begin{align*}
\text{'lab:inherit' (pred(t_1,\ldots,t_j),_?lb?,_?sf?).}
\end{align*}
\]

This extra clause is generated only if there is actually a definition for 'lab:inherit', i.e. if there are any normal class rules in the class template. This ensures that we do not leave any 'dangling predicates' in our translated program; but more importantly it also ensures that a local program does not incur any further overhead in the way of extra choice points.

The 'label:super' clause is already constructed for the use of super labelled conditions and for locally undefined conditions. We also need to allow a new call to a class template to access these inheritance clauses. We achieve this by adding a final 'link' clause to the set so far constructed:
Chapter 6: Implementing class templates

lab(_?pr?,_?lb?,_?sf?):-
  'lab:super'(_?pr?,_?lb?,_?sf?).

Since the lab:super clauses include code for both types of class rule, we must ensure that this link rule cannot be invoked in the case of a locally defined predicate. We can do this by adding a 'cut' to the other linking clauses:

lab(p(X₁,...,Xₙ),_?lb?,_?sf?):-
  !,
  'lab:p'(X₁,...,Xₙ,_?lb?,_?sf?).

This cut ensures that any labelled call which matches a link clause (and therefore is of a locally defined predicate) cannot go on to use the super link clause in the event of the locally defined program failing. Therefore such labelled calls will not use the overriding class rules.

On the other hand such labelled queries can use the normal class rules because of the inherit-link clause added to the end of the local programs. The cut in the link rule eliminates all the class rules, and the inherit clause 'puts back' the normal class rules.

Normally we would question the logical status of deliberately including cuts into our programs. However, in this case the intention is to produce efficient Prolog code. We have already seen that we can give a logically sound formulation of class templates. Furthermore, we can interpret the cuts as implementing an implicit form of negation — we could have put an extra condition in the super-link clause which filtered the labelled call for locally defined predicates.

We can now look at how a complete class template would be translated, for example the person program:

person(Ag):{
  age(Ag).
  likes(X):-X:age(A),A<10.
  likes(X):-X:like(self)
}.
person(Ag)<=animal.
person(Ag)<=system.
The list of clauses produced by the preprocessor from `person` is:

The local clauses for `age`:

'person:age'(Ag, _?lb?, _?sf?):-  
arg(1, _?lb?, Ag).

'person:age'(_1, _?lb?, _?sf?):-  
'person:inherit'(age(_1), _?lb?, _?sf?).

The local clauses for `likes`:

'person:likes'(X, _?lb?, _?sf?):-  
?:?(X, age(A), X),  
A<10.

'person:likes'(X, _?lb?, _?sf?):-  
?:?(X, like(_?sf?), X).  
% note the `self` keyword

'person:likes'(_1, _?lb?, _?sf?):-  
'person:inherit'(likes(_1), _?lb?, _?sf?).

The clauses linking the label predicate to the local predicates:

person(age(_1), _?lb?, _?sf?):-  
!,  
'person:age'(_1, _?lb?, _?sf?).

person(likes(_1), _?lb?, _?sf?):-  
!,  
'person:likes'(_1, _?lb?, _?sf?).

person(_?at?, _?lb?, _?sf?):-  
'person:super'(_?at?, _?lb?, _?sf?).

The super inheritance clauses:

'person:super'(_?at?, person(Ag), _?sf?):-  
life_form(_?at?, animal, _?sf?).

'person:super'(_?at?, person(Ag), _?sf?):-  
system(_?at?, system, _?sf?).

The `inherit` clause; note that there is no clause for `system`:

'person:inherit'(_?at?, person(Ag), _?sf?):-  
life_form(_?at?, life_form, _?sf?).
6.1.5 Equations and expressions

We saw in Chapter 4 that there are various ways in which we can incorporate equations and expressions into logic programming. One of these — the compilational approach — is particularly suitable in the context of a preprocessor since we are already preprocessing class templates into Prolog clauses.

We can (and do) integrate the two into a single compilation; this allows us to have equations embedded in class bodies and for class body axioms to contain expressions. In principle there is little difficulty in the combination. If there is an equation in the class body it is first compiled into a clause within the class body and then processed with the other clauses into Prolog clauses. Similarly, any expressions occurring in the class body axioms can be replaced by variables together with the conditions that give give them the correct value.

So, if we have a class template program such as:

```prolog
Hist: {  
- ~app([],X)=X.
  app([E|X],Y)=[E|app(X,Y)].
}
```

this is first translated into the standard class template:

```prolog
llist:{
  'app*'([],X,X).
  'app*'([E|X],Y,[E|_1]):-
    'app*'(_,Y,_).
}
```

and then into the standard Prolog clauses:

```prolog
'llist:app*'([],X,X,_?lb?,_?sf?).
'llist:app*'([E|X],Y,[E|_1],_?lb?,_?sf?):-
  'llist:app*'(_,Y,_1,_?lb?,_?sf?).
```

together with the corresponding linking clause:
§6.1 A preprocessor for translating class templates into Prolog

\[
\text{llist('app*(_1, _2, _3),_?lb?,_?sf?):-}
\]
\[
\quad '\text{llist:app*(_1, _2, _3),_?lb?,_?sf?)}
\]

One complication that can arise from the combination of class templates and equations is in the translation of higher-order features such as $\lambda$-expressions and set abstractions.

Suppose that we have a class template such as:

\[
\text{person(Ag):}
\]
\[
\quad \text{friendly} = \{ (X) \mid \text{likes(self)}, \text{likes(X)} \}.
\]
\[
\quad \text{likes(X)} : \text{-} \ldots \ldots
\]
\[
\}
\]

The friendly program is a \textit{relational}, that is the value of an expression such as \text{person}(30):\text{friendly} is a predicate which can be used to see if a given individual is friendly with this person or not. Such a value might not be applied immediately — the predicate might be stored in some list for later use; as in the query:

\[
\ldots, \text{map(lambda(P) *P:friendly, L)}, \ldots
\]

followed by a further sub-goal such as:

\[
\ldots, \text{Person on L, Person(john)}, \ldots
\]

in order to see if john is friendly with someone on the list L.

If we take the same approach to translating friendly as we would for first order equations then we would translate the person template as follows:

\[
\text{person(Ag):}
\]
\[
\quad \text{'friendly*('person:0,3).}
\]
\[
\quad \text{'person:0'(X):- likes(X), X:likes(self).}
\]
\[
\quad \ldots\ldots
\]
\[
\}
\]

which, when further transformed into Prolog clauses gives us:

\[
\text{Our preprocessor generates symbols of the form 'label:X' rather than $\lambda$-symbols.}
\]

3
Chapter 6: Implementing class templates

'person:friendly*'('person:0',_?lb?,_?sf?).
'person:person:0'(X,_?lb?,_?sf?):-
  'person:likes'(X,_?lb?,_?sf?),
  '?:?'(likes(_?sf?),X,X).

with a linking clause:

person('person:0'(X),_?lb?,_?sf?):-
  'person:person:0'(X,_?lb?,_?sf?).

The problem with this translation is that when the set abstraction is eventually applied, as in:

...,Person on L, Person(john),...

the original context may have been lost. Under our mapping this pair of conditions — in the case that they actually occur in a different class template, directory (say) — would become:

..., 'directory:on'(Person,L,_?lb?,_?sf?),
  '?apply?'(Person,[john],_?lb?,_?sf?),...

The '?apply?' program has a similar role to the '?:?' program, except that it relates to variable predicates as opposed to variable labels. We can define it as follows:

'?apply?'(Pred, Args, Label, Self):-
  functor(Label, L, _),
  Atom =..[Pred|Args],
  L(Atom, Label, Self).

The effect of applying the value of friendly (and therefore 'person:0') would eventually be to form the sub-goal

..., 'person:0'(john, directory, self),...

since the actual call to the set abstraction took place within the directory class template. The problem with this is, of course, that 'person:0' is defined in the
§6.1 A preprocessor for translating class templates into Prolog

person template not in the directory template. Furthermore it refers to values for its 
_?lb? and _?sf? arguments that were valid at the time that friendly is evaluated rather than when the value of friendly is applied.

The true cause of the problem is that we did not include as free variables in the definition of 'person:0' the '_?lb?' and '_?sf?' variables. In fact these variables should be considered free in every lambda expression and set abstraction that appears in a class template. If we do include these variables as free variables, then we get a slightly different translation of friendly:

'person:friendly*'( 'person:0' ( _?lb?, _?sf?), _?lb?, _?sf?).

We must alter somewhat the definition of ?apply?. When we apply a lambda expression or set abstraction the extra free variables corresponding to the label and self arguments which are in the lambda term are used instead of the label and self arguments that are valid at the time of the 'apply' call. Our definition of 'apply' becomes:

'?apply?'(P, Args, Label, Self):-
    atom(P),!,
    functor(Label, L, _),
    Atom =..[P|Args],
    L(Atom, Label, Self).

'?apply?'(P, Args, _, _):-
    P =.. [Pr, Label, Self|F],% decompose predicate
    NP =.. [Pr|F],% reform link symbol
    Atom =.. [NP |Args],% reform link predication
    functor(Label, Lb, _),% determine actual class
    Lb(Atom, Label, Self).% invoke actual class

With this translation, the application of the value of friendly would look like a goal such as:

..., '?apply?' ('person:0'(person(32),tom),
    'person:friendly*'( 'person:0'( _?lb?, _?sf?), _?lb?, _?sf?).

which — after a number of uses of the 'univ' operator =.. — would be mapped to the call for person:
Chapter 6: Implementing class templates

To complete the connection with the locally defined predicate the linking clause for `person:0` within the `person` program would be invoked to give us the new call:

```
..., 'person:person:0'(john,person(32),tom),...
```

Since the only way that a lambda expression or set abstraction can be invoked is via the use of a variable function (or variable predicate) and since such calls are mapped to calls to the `?apply?` primitive we can actually short-circuit this last step and include the clause(s) for lambda expressions and set abstractions directly in the definition for `person`:

```
person('person:0',X,_?lb?,_?sf?):-
    'person:likes'(X,_?lb?,_?sf?),
    '?'(likes(_?sf?),X,X).
```

This saves one inference step for each application of a lambda expression. In a realistic implementation of the preprocessor it may also be advisable to optimize the implementation of `?apply?` with low-level support functions: it requires three applications of the `-...` primitive and one call to `functor` in order to make a single call involving a predicate or function variable.

### 6.1.6 Named functions

The reader will recall that the primary function of lambda expressions and set abstractions is to allow ‘nameless’ functions and predicates to be specified as arguments to operators such as `map` and `reduce`. It is just as important to be able to use named functions and predicates with these operators. However, this implies that symbols may occur in the text of a program where they appear to be constant symbols when in fact they are predicate symbols and function symbols.

In general it is not possible — in an untyped language — to determine is a given constant symbol is really a predicate symbol (or a function symbol) in disguise rather than a normal constant symbol.

In the case of function symbols this is an especially difficult issue since when functions are translated into relations the function symbol itself is modified — by
adding a * character to the end — to form a new predicate symbol. The effect of this is that when a named function is eventually applied (by means of a variable function symbol) it may not be possible to recover the correct definition. In any case it may require the dynamic construction of predicate symbols from function symbols; this in turn can be quite expensive since it might involve searching a symbol table.

Since the compiler cannot, in general, detect when a constant symbol is really a function symbol we propose that the programmer be required to identify when a constant symbol in the program is actually the name of a function; as, for example, in the expression:

\[ \ldots \text{reduce}(\text{sum}, L) \ldots \]

The most obvious way to do this would be to append a * to the end of the name:

\[ \ldots \text{reduce}(\text{sum*}, L) \ldots \]

The * is a (post-fix) function whose value is the predicate symbol corresponding to its argument: the value of \( \text{sum*} \) is the constant symbol \( \text{sum*} \). It is not actually required for the argument of * to be known at compile-time; however if it is then the pre-processor will evaluate * directly rather than leave it to dynamic execution.

In fact, * has to be a little more sophisticated since, for reasons similar to those involved in handling \( \lambda \)-expressions and set abstractions, the correct value of * is not simply a constant but a compound term which mentions the \'\_\?!lb\?' and \'\_\?!sf\?' variables. So the correct value of \( \text{sum*} \) is the term \( \text{sum*} (\_\?!lb?, \_\?!sf?) \). When such function terms are applied to arguments — via the '?apply? or ':? primitives — then the \'\_\?!lb\?' and \'\_\?!sf\?' arguments in the \( \text{sum*} \) term are used instead of those current at the time of the application of the function.

\[ 6.1.7 \text{ Object search} \]

In the implementation strategy that we have outlined so far we have assumed that we know at the time of any call both the label (in the case of a labelled call) and the predicate symbol. This allows us to use efficient indexing techniques to search for the code of a class template and for a program within a class template.
However there remains the possibility that we cannot identify the label or predicate symbol at the time of a call. In a production compiler system we must be able to handle this situation in a graceful manner. We could, for example, decide simply to report an error. This is not altogether satisfactory since there is a legitimate use of the variable label as a form of object search.

On the other hand, neither do we want to implement class template programs simply by compiling them into class clauses as we outlined in Chapter 5. This would allow full object search but at an unacceptable cost in overall performance when not searching for objects.

Another approach would be to invoke a kind of label error handler in the situation where the label is unknown at the time of the call. For example, in the call:

\[ \ldots, X: \text{colour}(\text{red}), \ldots \]

the system could dynamically replace the label \( X \) to get the call:

\[ \ldots, \text {?object?}(X): \text{colour}(\text{red}), \ldots \]

if \( X \) was unbound. (The exact choice of the \text {?object?} label is not important of course.) The programmer (or even the compiler) could provide a definition of the \text {?object?} label via a set of class rules, one for each label in the program:

\[
\begin{align*}
?\text{object?}(\text{label}_1) & \equiv \text{label}_1, \\
\ldots \\
?\text{object?}(\text{label}_n) & \equiv \text{label}_n.
\end{align*}
\]

The effect of this is to provide a search through all the objects identified by \( \text{label}_1 \) through \( \text{label}_n \) and on a successful completion the variable \( X \) would be bound to a label term for which the \text{colour}(\text{red}) condition was true. (I.e. find an object which is coloured red.) Backtracking could be used to search for alternative labels as necessary.

A similar technique can be used to handle the variable predication case where the predication or predicate symbol was unknown at the time of a call.
6.1.8 Representing class template programs

The pre-processor is a meta-program: it takes as arguments the name of a class template program and returns the name of a 'normal' Prolog program. This latter term would then be passed to a standard Prolog compiler for further compilation.

We use a particular scheme for representing object-level formulae in the meta-level: this is the $\Sigma$-scheme. (This same scheme is also used in MacProlog although this fact is hidden from the casual programmer.)

In the $\Sigma$-scheme a formula is represented as a ' - ' pair consisting of a list of variables (or rather a list of the names of variables) and a term representing the formula itself. In this term connectives such as ',', ' : ' are represented by terms with the same name. Predications are represented by compound terms of the same arity as the predication and whose function symbol is the same as the predicate symbol of the predication. Compound terms are also represented by themselves as are constants and numbers.

A variable is represented by a constant in the $\Sigma$-scheme. We distinguish between constants naming constants and constants naming variables by the fact that the latter also appear in the list of variables. So, a clause such as:

\[
\text{app}(\text{[E|X]}, Y, [E|Z]) :- \text{app}(X, Y, Z)
\]

would be named by the term:

\[
[\text{'E'}, \text{'X'}, \text{'Y'}, \text{'Z'}] - (\text{app}([\text{'E'|'X'}], \text{'Y'}, [\text{'E'|'Z'}]))
\]

One benefit of this naming scheme (apart from its logical soundness of course) is that it becomes possible to remember names of variables in clauses. This is something that few Prolog systems achieve. A further advantage of the $\Sigma$-scheme is that we can use variables which would not normally be interpreted as variables in the standard Edinburgh Prolog syntax. For example, the \text{app} clause could just as easily be named by the term:

\[
[\text{e}, \text{x}, \text{y}, \text{z}] - (\text{app}([\text{e}|\text{x}], \text{y}, [\text{e}|\text{z}]):-\text{app}(\text{x}, \text{y}, \text{z}))
\]

We used this property when we introduced the special variables '_?1b_?' and
"_?sf?" in our preprocessor to represent the label and self arguments.

A class template can be represented by a similar kind of term to the clause, except that we now have nested formulae represented by nested ' - ' pairs within class bodies. Each axiom within the class body is also a pair of variables and formula:

\[
[ 'Ag' ] - (\text{person}( 'Ag' ) : \{
  [ ] - \text{age}( 'Ag' ) .
  [ 'X', 'A' ] - (\text{likes}( 'X' ) : -
    'X' : \text{age}( 'A' ), 'A' < 10 ).
  [ 'X' ] - (\text{likes}( 'X' ) : - 'X' : \text{like}( \text{self} ) )
}\}
\]

We have made a few changes to the standard term syntax in regular Prolog. The normal \textit{dot-space} terminator used to signal the end of a term in Prolog is reinterpreted if it occurs within a class body; i.e. if it occurs within a pair of \{ \} 's. In this case the \textit{dot-space} terminator is interpreted as the symbol ' . ' which is also an operator of priority 12014. We have also adjusted slightly the expected precedence of the term inside \{ \} 's to be 1201 also. This allows us to have terms which represent class bodies without having to force the Prolog programmer to use a different symbol for the end of a clause.

Other changes include the possibility of function symbols of compound terms to be variables or themselves compound; and the ability to have variable predicate symbols. All of these changes have the property that they strictly extend the term syntax of Prolog: the new features would be syntax errors in the standard syntax, and a terms of standard Prolog is the same term in the extended syntax.

The preprocessor program itself expects a class template in the form described above, and it returns a list of clauses; again represented using the \Sigma-scheme. Some of the clauses, in particular those arising from lambda expressions will have complex predicate symbols in them. If required, the list can be further processed to remove those 'funny' predicate symbols.

\footnote{The maximum priority of a term in Edinburgh Prolog syntax is normally 1200.}
6.1.9 A comparison of class template programs with Prolog programs

The mapping that we have described from class templates to Prolog programs produces programs which have very little overhead compared to the original programs. If we compare a Prolog program with the translated class template then the primary overhead amounts to two extra arguments in every condition. This will reduce the perceived performance of a program these two arguments have to be allocated, passed as parameters and so on.

The other potential sources of overhead are extra choice points (where none were indicated from the Prolog program), extra resolution steps and extra conditions in clauses and a higher memory turnover (i.e. greater amount of garbage needing to be collected).

The local programs have the same number of clauses as the Prolog equivalents except in the case where the class template has a normal class rule. In this case an extra choice point will be needed in case the local program fails and an inherited version is needed to solve the goal. This amounts to an extra overhead, especially in the case where there are no other definitions anyway.

A global analysis of a complete class template program might, by determining which programs are defined in which class templates, be able to determine that a given local program has no alternatives in the class templates which are accessible. In this case then the extra inherit linking clause would not be needed and therefore could be omitted. However, such a global analysis would only be worth while where the complete program was not evolving since a minor change could mean the re-compilation of the whole. Furthermore, if the programmer has only used overriding class rules then the extra choice point would not be generated in any case thus providing a simple remedy.

Extra resolution steps are introduced into the computation whenever an explicitly labelled call is made and whenever class rules are invoked. A labelled call is compiled into a call to the 'label predicate'; a linking clause for the label predicate establishes an entry point into the 'local predicate' that was actually invoked. The overhead for a labelled call where the label is a variable rather than determinable at compile is much higher: the '?' program involves the use of three calls to '=' which may be quite expensive in some Prolog systems. The efficient implementation of '?' is likely to be quite beneficial more better performance in
class template programs.

An inheritance step is likely to involve up to three inference steps depending on the situation. If a local predicate 'backtracks' its way into using a class rule then there are three steps involved before the next local predicate may be tried: one step to access the local class rules, one step to perform the inheritance step and finally the linking rule in the new class template. This overhead can be reduced at the cost of larger generated code by replicating the class rules with each local predicate for example.

Extra conditions are added to the program when a clause references a label argument — a call to the system built-in primitive arg — and when a condition has an evaluable function as an argument. Either of these could, in principle, weaken one of the standard optimizations of Prolog compilers where the first condition in a clause is treated specially. If the new condition is added as the first condition is a clause then the programmer's first condition is no longer optimized in the same way. The importance of this depends on the Prolog compiler itself. Some compilers recognize some system predicates (possibly including arg) and deal with them in a special way. In this case the original optimization for the first call in a clause may apply anyway.

A class template program has a similar memory usage profile to a normal Prolog program except where labelled calls are used. In this case copies of label terms are made at the point of an original labelled call, and whenever a class rule is used for inheritance. If an application makes heavy use of inheritance then this might become a significant factor. However since this is an 'extra', and is not incurred in great measure by normal Prolog programs this impacts more on 'native' class template applications than on Prolog applications.

Overall, we feel that the overhead of using class templates is quite small. Preliminary measurements suggest an overhead of some 5-15% compared with normal Prolog programs. In the following section we shall see how one would sketch a direct compiler for class templates which would eliminate the overheads completely.
§6.2 Compiling class template programs

As well as building a preprocessor for class templates it is also useful to see how we might modify a standard Prolog compiler to compile class templates directly. Such a system would eliminate the (admittedly small) overheads involved in preprocessing; however, such a system would also allow some class template applications to actually be more efficient than their Prolog counterpart. This is primarily due to the savings which arise from the fact that label variables would not be arguments in every clause.

The reader is assumed to be familiar with basic Prolog compiler technology. For a detailed presentation of current compiler techniques the reader is referred to [Warren'83], [Gabriel et al'86] and [McCabe'83,84].

A class template program bears a strong resemblance to a module, albeit a module with parameters and inheritance. Our strategy for compiling class templates recognizes this and takes advantage, primarily on the same assumption that we made for the preprocessor that a large proportion of computations are within class bodies rather than between class templates.

To see how a class template may be compiled we consider the sequence of actions our modified Prolog compiler system must go through to call a program inside a given class template:

\[ \ldots, \text{person}(33) \text{self:likes(john)}, \ldots \]

In a call such as this there are two sets of arguments (as opposed to one with the standard Prolog compiler) and a self reference to build before we can enter the likes program: the label arguments and the predicate arguments. Since the label arguments and the self label are constant across the whole of the called class template (in much the same way that the non-label variables in a clause are constant across that clause) we have to represent them in such a way that they can be accessed (i.e. unified with) during the evaluation of likes. We do this by constructing a global environment vector for all the label parameters and for the self label. The arguments to the call itself may be placed in argument registers in the same way that a regular Prolog compiler does.

Since the interpretation of a given predicate symbol is dynamic (it will be different
in different class templates) we can no longer rely on a single dictionary structure to 'magically' give us the compiled code for likes. Instead we construct a local dictionary in each class template and on entry to the class template we search this local dictionary for the appropriate code. The predicate symbol of the call is passed in a special register (it could be a designated argument register) to enable this search. Having constructed the arguments to the call and having constructed a call record if necessary we can now enter the class template proper.

On entering the person program we must decide which (if any) of the compiled clauses from within person to use to solve the likes goal. We do this by means of an indexed search on the likes symbol in the list of locally defined predicates. This search is similar in principle to the indexed search on the first argument of a call which is commonly used to give improved access to clauses in large Prolog programs. The efficiency of the search for the code associated with a predicate symbol within a class body determines to some extent the overall performance of the system: we do not wish inter-class calls to be slow since that would detract from the overall utility of the system.

The search is only invoked on the initial entry of a labelled call into a class template. Once 'inside' the class template it is not usually necessary to invoke the search again.

Structure of the compiled person class template

```
person(A): Establish global environment (A), and self register
switch on predicate symbol of call:
  case likes: Try to solve like with
               local code, else try the
               class rules
    case age:
    default: Try first class rule...
             Try second class rule...
             ...

  Taken on failure of
  local definition(s)
```

Structure of the compiled person class template
§6.2 Compiling class template programs

We can optimize the search by using a hash table to represent the defined predicate symbols within a class body and utilizing some pre-hashing techniques: we can associate with each constant symbol a pre-hash value which might consist of some function of the length of the symbol and the characters in its print name. The indexed search may be implemented via a hash coding of the pre-hash values rather than on the full print name.

Once the compiled clauses for a predicate symbol have been determined, execution continues in the normal Prolog style, trying each of the applicable clauses in turn until one succeeds to solve the original goal. We can also apply the standard argument indexing techniques within the clauses for each predicate within the class body.

Individual clauses of a sub-program can be compiled in the same manner as in a conventional Prolog compiler; the main exceptions relate to inter-class calls (which we have already covered), label variables and self references.

Normally, on entry to a Prolog clause an environment vector is established which has an entry for each of the local variables in the clause. Subsequent unification and procedure calls may refer to the local environment (using offsets generated by the compiler) to instantiate or read the values of these variables. In our case we have two kinds of variables: local or clause variables and global or label variables. We must take this extra classification into account when we compile instructions which refer to variables. This is relatively simple for the compiler to do since the distinction between the two kinds of variable can be completely determined at compile time.

In fact this separation of variables into local and global is a source of a small improvement over regular Prolog; since the global environment will not change whilst execution is within a class template the values of those variables do not need to be passed between calls or otherwise referred to except when actually needed inside an individual clause. This saves time and space compared to the equivalent situation with a regular Prolog program.

Any occurrences of the self keyword are compiled into instructions which access the term contained in the self register. If the self keyword was in the head of the clause then the corresponding argument may be unified with the self term. Usually however, the self term will be used in constructing an argument to a call in the body of an axiom.
Having entered a clause and (presumably) successfully unified the actual arguments of the call with the head of the clause, the body of the clause is executed. In a normal Prolog compiler the body of a clause is compiled into code which is solely concerned with constructing calls and executing them. With our system we have a similar situation except that there are now two types of call: the local or unlabelled call and the labelled call. We have already seen how a labelled call is compiled; the unlabelled call is compiled in exactly the same way as in Prolog.

When it comes to determining the address of the compiled clauses for the locally defined predicate we have a second advantage over conventional Prolog: we can resolve at compile time (or more likely at link time) the addresses of all the programs within a class template. This enables us to short-circuit many aspects of a normal call such as searching in the class template predicate dictionary and checking the number of arguments.

There are in fact three types of compiled clauses which may appear in a compiled class template program: the code for the locally defined clauses from the class body, the code for the various class rules associated with the class and clauses which have been dynamically asserted.

### 6.2.1 Choice point and call record structures

When a clause is attempted for which there are alternatives then a backtracking choice point is established; similarly if a goal is called and there are more goals to follow in the same clause then a call record is established. The principle behind these data structures is the same for each: sufficient control information must be preserved in the relevant structure so that computation may continue subsequent to the failed clause (or succeeded goal) as though the attempted clause (or goal) had not existed.

For the called goal in a regular Prolog system this means that each call record must represent the following control information: the current local environment, the next goal to attempt and the parent call record (where to succeed to should the current clause finish successfully).

In the case of class templates we have more to remember: the global environment as well as the local environment and the self control register. So, a call record can
be represented with the structure:

```
Parent call record
  Local environment
  Next goal
  Global environment
  Self label
```

Regular call record

```
Extended call record
```

Call record data structure

Also helpful, though not essential, to enable a simple implementation of cut is the current choice point which was active on initial entry to the current program. This appears in both forms of the call record.

The choice point structure is commonly split into two separate parts: the choice point proper and the reset list or trail. Since the main function of a choice point structure is to preserve the state of the system to enable the next clause to be tried on backtracking a choice point record needs to hold the following control information in a regular Prolog system: the arguments of the current goal, the current call record, the next clause to try, the trail of bound variables and the previous choice point. In an extended record we also need to record the predicate symbol of the current goal (this will be needed if a class rule is to be attempted) and references to the global environment and the self label.

```
Arguments to call
  ...
  Previous choice point
  Current goal record
  Next clause
  Trail
```

Regular choice point

```
Predicate symbol
  Global environment
  Self label
```

Extended choice point

Choice point data structure

As we can see, the extended structures are not too different to the standard structures. This suggests that the space requirements for executing class template programs are not onerous.
6.2.2 Class rules

A class rule can be compiled in a very similar manner to a regular clause, however its body consists of a single inter-class call. In fact only half of the inter-class call need be constructed by virtue of the fact that this is a class rule and the predicate part of the call and the self label will already have been constructed. The main point of interest is the way that the class rule code is linked in with the rest of the code for the class template.

Recall that on entry to a class template program we search the local dictionary to see if there are any clauses that might be used to solve the condition. If the local search is successful then the compiled code for those local axioms is attempted via 'try' instructions in the normal manner. However, if this search fails then the various class rules associated with the label must be tried. The compiled code for the class rules are collected together into the default part of the switch_on_predicate statement. This applies both to normal class rules and to overriding class rules.

Backtracking can be used to explore the alternative class rules in exactly the same way that backtracking is used to explore the alternative clauses in a standard Prolog program.

An added complication arises when the local axioms have all been explored (unsuccessfully). In this situation normal class rules are also tried to see if another class template program can be used to solve the condition. Overriding class rules are not attempted when backtracking from normal axioms since by definition a local definition of a relation overrides any inherited definitions.

We can implement this extra backtracking possibility by linking in to the compiled code for the local axioms the compiled code for the normal class rules. In fact this can be done almost as though the the class rules were extra clauses for each of the locally defined relations.

So, for example, if we have a class template for penguins such as:
penguin:
{
    mode(swim).
    habitat(south_pole).
}.
penguin<<bird.
penguin<=publisher.

then the compiled form of this showing how the various types of code are linked together is:

```
penguin: switch_on_predicate
    case 'mode':
        try
            try_class_rule
                mode(swim)
    case 'habitat':
        try
            try_class_rule
                habitat(south_pole)
    ...
    default:
        try
            penguin<<bird
            penguin<=publisher
```

The try_class_rule instruction is actually the same as the normal try instruction used to attempt individual clauses. We have given it a different name to emphasize the split between normal axioms and class rules.

Notice that the test that is necessary, in the case of an overriding class rule, to filter out locally defined predicates is essentially performed at compile time. If we actually had to filter out these predicates at run-time then it would be rather expensive to use an overriding class rule. This is especially true if a class template had many local axioms.

If a condition has a super label then we must compile the call so that the local axioms are not attempted. The simplest way of doing this is to route all super calls directly to the default clause in the switch_on_predicate statement. This automatically bypasses any local definitions.
One complication that can arise with compiling super relates to overriding class rules. If a condition is super labelled, and the predicate of the condition does have local axioms then the overriding class rules must not be attempted. However, since this is a condition which can be detected at compile time we can generate special code for the situation. We do this by having a duplicate of the default class rule code sequence which omits all references to overriding class rules. The super calls are directed to this section of code rather than the default section. (In practice we do not imagine that the combination will occur very frequently.)

6.2.3 Dynamically asserted clauses

Any dynamically asserted clauses\(^5\) are compiled in much the same way as other clauses in a class body. There are however some complications due to the fact that the label may not be 'pure' in a dynamically asserted clause: it may contain non-variable terms. This implies that the unification code for such a clause has to unify with the global environment as well as the arguments to the call.

There also remains the problem of where in the overall sequence of clauses such dynamically asserted code should be inserted. We suggest the appropriate place is between the locally defined clauses and the class rules (assuming of course that class rules are always attempted after any locally defined clauses). Because of the complexities which can arise when manipulating the code which is associated with the original class body we suggest limiting dynamic clauses to new predicates not already defined in the class body.

§6.3 Summary

In this chapter we have not attempted to provide a definitive description of the implementation of class template programs. Instead we have concentrated on a preprocessor for class templates into Prolog. The performance of this preprocessor is quite adequate; however the examination of a native class template system is worth pursuing also. This is especially true if a complete programming system is built from class templates. The differences between a regular Prolog compiler and our system are quite manageable.

\(^5\) We do not pass judgment on the use of dynamically asserted clauses except to note that any form of dynamic assertion is expensive. Here we indicate a possible implementation strategy.
Conclusion and future work

The main contribution of this work is in the establishment of a relationship between object oriented programming and logic programming. We have seen, with the class template language, that the program structuring and knowledge representational features of object oriented programming can be incorporated into logic programming without compromising the framework of logic programming.

The manner of the integration of the class template language also represents an interesting example of how a language can be ‘layered’ on top of logic programming and still inherit the semantic properties of logic.

The examples of using the class template language that we have explored suggest a number of programming techniques not commonly found in logic programming practice. First among these is probably the ‘data-driven programming’ technique. The hallmark of data-driven programming is the way in which the form of the data to be evaluated/processed identifies the programs needed to perform the evaluation.

In the arithmetic expression evaluation example, the data consists of terms such as “+(1,2)” and “*(3,+(1,2))” to denote expressions. Such expressions are evaluated by invoking programs within the “+” and “*” class templates. The particular feature of data-driven programming in class templates is that the label of a condition is used to represent both the data to be evaluated and the programs used to evaluate it.

A more impressive application of the data-driven programming technique is described in Chapter 3 on denotational graphics. In that chapter we used terms to denote pictures (in an analogous way that we use numerals such as 2165 etc to denote numbers). Graphical computations such as computing drawing sequences and determining if a point is inside a picture are expressed in terms of ‘sending messages to pictures’. Class templates provide an ideal language for describing pictures and the relations and functions over picture terms. Furthermore, the class template language also provides a basis for describing graphical databases and graphical applications.

The graphics system described in Chapter 3 is actually a ‘rational reconstruction’ of the graphics language and system implemented in MacProlog. This system offers
significant benefits to the application programmer in terms of faster programming of graphical applications. We would hope that a graphics system which is actually based on class templates would also offer similar advantages.

Apart from some novel programming techniques the class template language also shows how we might address one of the harder problems in practical programming — namely the design and construction of large modular systems. Modules are important in logic programming because logic programs, like programs in other languages, can grow to be large. In fact the real benefit of a high-level programming language such as Prolog must be that it makes programming easier and better. This is even more important for large programs than for small ones since the sheer scale of large programs makes building them difficult. On the other hand standard logic programming notations are too fine-grained to be well suited for large projects.

Viewed as a language for modules the class template language offers a number of advantages over other Prolog-based module systems. Firstly, and most importantly, it allows modules to have a logical status. A class template module is an atomic theory — a set of relations.

Many module systems are based on partitions of the symbol space. Some module systems partition the whole symbol space — as in micro-Prolog for example — while others partition the predicate symbol space only — for example Quintus Prolog. Such a view of modules as partitions is really orthogonal to the standard logical perspective: a partition of the symbol space does not relate to the predicates defined over that space.

The class template language provides other advantages than simply giving modules a declarative semantics. For example the notion of a generic module arises naturally out of the fact that a class label can have parameters; and that label terms can be arguments to other label terms. We saw this in the simple sort module where the particular ordering to be used in sorting lists was specified as a parameter of the simple class template.

We can also construct composite modules: we built a composite ordering from a generic ordering over pairs of terms and used that to allow us to sort lists of pairs without having to modify in any way the original simple program. Such generic modules are not available in many programming languages; although some of the
The same benefits can be found in strongly typed functional languages such as ML.

A further benefit of the class template language is that we can use class rules to structure complex modules. For example, if we wished to implement a complex library or an environment in which user programs are to be embedded, we can use the class rules to make this easier. All that is needed for a user's program to have access to a library is the inclusion of a class rule with the appropriate templates:

\[ \text{label} \Leftarrow \text{library} \]

Such a class rule might be explicitly written by the programmer or it may even be automatically constructed by the system for certain standard libraries. When it comes to the implementation of a library it may be desirable to structure the library as a collection of sub-libraries. For example the library may contain a graphics sub-library and a file-system sub-library. Such a division can be described by defining the library class template as a set of class rules: one class rule for each part of the library:

\[
\begin{align*}
\text{library} & \Leftarrow \text{graphics	extunderscore lib}. \\
\text{library} & \Leftarrow \text{file	extunderscore lib}.
\end{align*}
\]

This structuring of the library would be invisible to the user of the library; it simply aids the library's construction. Such a means of specifying modules is an especially powerful method for building large and complex systems.

The class template language was implemented by means of the preprocessor described in Chapter 6. This is a reasonable basis for implementing them on a more-or-less standard Prolog system. The preprocessor does however use a number of features which are specific to MacProlog; these make the burden of implementing class templates somewhat less than it might be in a 'standard' Prolog system.

For example, in MacProlog it is possible to have variable function and predicate symbols; we can also have compound function symbols. This makes the representation of higher order expressions much simpler. Furthermore the natural representation of programs in MacProlog is the same \( \Sigma \)-model\(^1\) that we have used to name class template programs in the preprocessor. This is a more logically

---

\(^1\) Although the standard Prolog representation is also supported in MacProlog.
consistent method of naming clauses and programs.

The primary purpose of the construction of the preprocessor was to demonstrate the feasibility of layering the new language on top of Prolog. In this aim the preprocessor was successful; furthermore, it demonstrates that class template programs do not have to be much slower than normal Prolog programs. In the future it may be better to build support for class templates at a lower level than is possible via the preprocessor.

The greatest challenge yet to be addressed in the development of the class template language is undoubtedly the representation of state in evolving systems. We saw in Chapter 2 how this issue has been tackled by modifying the standard proof procedure. We feel that this is too drastic an approach and throws away too many of the properties of logic programming, in particular it is hard to arrive at sensible declarative semantics in such systems.

Our preferred alternative is to employ a metalogical approach to separate out the aspects of change in an evolving system from the declarative aspects of the system’s components. The model here is of a collection of cooperating experts — each expert is modelled as a collection of class templates. If a given expert is asked a question the standard proof procedure will deliver a declaratively justifiable answer. The cooperation between the various experts and the evolution of their knowledge is modelled as a metalogical process of evolving sets of axioms. In such a system the object level represents the declaratively justifiable and the meta-level ‘contains’ the change.

By separating out change from proof and truth we certainly make the handling of evolving systems easier. However this approach may not be sufficient to model all evolving systems since some such systems change themselves. Furthermore, there may also be issues of efficient implementation of combined meta-level/object-level systems.

Apart from the problems of modelling change a second major area to be investigated is that of methodology. In particular a methodology for building large-scale systems. One approach to such a methodology be be to liken processes such as step-wise refinement and requirements analysis to the progressive description and refinement of modules using inheritance. However this is an area which is still to be pursued.
Appendix A: Syntax of class template programs

The notation we use is a variation of definite clause grammar notation; the differences being the incorporation of optional productions (denoted by «optional» which represents an optional occurrence of optional) and repeated productions (denoted by repeated* which represents 0 or more occurrences of repeated, and repeated+ which represented one or more occurrences).

The syntax of class templates programs is based on 'standard' Edinburgh Prolog syntax as described in [Clocksin & Mellish'84]. However we have taken the liberty to clean up some of the loose ends in that notation and of course there are a number of significant extensions. We retain, however, the basic operator precedence grammar.

program --> class_template*.

query --> expression, "?".

class_template --> «class_body», class_rule*.

class_body --> constrained_label, ":", ":", axiom*, ":".

class_rule --> constrained_label, ("<=" | "<<"), label.

axiom --> clause | equation.

clause --> head_predication, «"-"", body».

equation --> l_h_s, "=", expression, «"-"", body».

l_h_s --> constrained_term.

head_predication --> constrained_term.

class_template --> constrained_label --> constrained_term.
Appendix A: Syntax of class template programs

body \rightarrow \text{disjunction} | \text{conjunction}.

disjunction \rightarrow \text{conjunction}, ";", \text{disjunction}.

conjunction \rightarrow (\text{negation} | \text{condition}),
\text{"","}, \text{conjunction}.

negation \rightarrow "\neg" | "\vdash", \text{condition}.

condition \rightarrow «\text{label}, \text{":"}, (\text{predication}|"(", \text{body}, ")")

label \rightarrow \text{term}.

predication \rightarrow \text{expression}.

expression \rightarrow \text{term} | \text{lambda} | \text{set_abstraction}.

term \rightarrow \text{term}(1000)
\text{"\\`"}, \text{expression}.

constrained_term \rightarrow \text{term}, \{\text{not a list, variable, number or string}\}.

term(N) \rightarrow \text{prefix\_op}(O,R), \text{term}(R), \{0 \leq N\}
\text{|} \text{term}(L), \text{infix\_op}(L,O,R), \text{term}(R), \{0 \leq N\}
\text{|} \text{term}(L), \text{postfix\_op}(L,O), \{0 \leq N\}
\text{|} \text{term0}.

term0 \rightarrow \text{simple\_term} | \text{term0}, \text{arg\_list}.

simple\_term \rightarrow \text{variable}
\text{|} \text{constant}
\text{|} \text{number}
\text{|} "(" , \text{term } ")"
\text{|} \text{list}
\text{|} \text{string}
\text{|} \text{bracket\_term}.
Appendix A: Syntax of class template programs

list -> "[]
     | "[", term, (",", term)*, "\"]", term", "].

bracket_term --> "\{", term, "}".

arg_list --> "(", term, (",", term)*, ")".

lambda --> "lambda", "\ •\", expression, "\":-\", body".

set_abstraction --> "\{", arg_list, "\|", body, "}".

variable --> (uppercase | "\_"), alphanumeric*.

constant --> identifier | quoted.

number --> "\"-\", digits, "\", digits,
          "\"(\"e\"|\"E\")", "\"-\", digits\"".

identifier --> lowercase, alphanumeric* | graphics.

quoted --> "\"", any_char*, "\"".

string --> "\"\"", any_char*, "\"\"".

alphanumeric --> lowercase | uppercase | digit | "\_".

digits --> digit*.

digit --> "0" | ... | "9".

graphics --> graphic*.

graphic --> any_char except digit, lowercase, uppercase, punctuation.

lowercase --> "a" | ... | "z".

uppercase --> "A" | ... | "Z".
Appendix B: Class template preprocessor

This program is a MacProlog program which forms the core of a simple class template front end for MacProlog.

Preprocessor for Logic and Objects
F.G. McCabe

converts a complete class template program into 'standard' clauses.

/* first some useful operator declarations */
:-op(701,xfx,':'). % labelled call operator
:-op(1201,xfy,'.''). % clause separator in a class body.
:-op(1200,xfx,'<='}). % class rule operator
:-op(698,fx,'\'). % back-quote used for escaping interpretation
:-op(698,fx,\). % anti-quote operator
:-op(699,xfx,\). % used in lambda expressions
:-op(999,xfy,'|'). % denotes a union set abstraction
:-op(400,yf,'*'). % function symbol indicator

/* translate a complete class body */
tr_template(LbL,Body,Rules, Rls, Rs):-
   tr_rules(Rules, Ri, Rs, Inherit, Super, LbL),
   tr_class_body(Body, Rls, Ri, Preds, Funs, Inherit, Super).

/* Generate the clauses for a class body. This involves sorting the clauses in a class body according to predicate/arities. */
tr_class_body(LabVars-(Label:{Clauses}),[Lb(EntryPs)|R], Rs, Preds,Funs,In,Su):-
   sort_sentences(Clauses,Relations,Preds,Functions,Funs),
Appendix B: Class template preprocessor

termin(Preds),
termin(Funs),
Label=..[Lb|LbArgs],
tr_methods(Relations, 0(Ax), Ay, Ri, (Funs,Preds),
\[Lb(LbArgs),In]),
tr_functions(Functions, Ay, _\[Az], Ri, Rs, (Funs,Preds),
\[Lb(LbArgs)],
mk_entry_rels(Preds,Lb,EntryPs,Ei),
mk_entry_funs(Funs, Lb, Ei, Ax),
mk_super_rule(Lb, Az, [], Su),!.

sort_sentences([], Relations, _, Functions, _).
sort_sentences([Cl|Clauses], Relations, Preds, Functions, Funs):-!,
\[sort_sentence(Cl,Relations,Preds,Functions,Funs),
\[sort_sentences(Clauses,Relations,Preds,Functions,Funs)).

sort_sentence(Cl, Relations, Preds, Functions, Funs):-
defining_fun(Cl,F,A),!, % is this an equation?
occ(F/A,Funs),
occ(eqs(F,A,Eqs),Functions),
occ(Cl,Eqs).

sort_sentence(Cl, Relations, Preds, Functions, Funs):-
defining_pred(Cl,Pred,Arity),
occ(Pred/Arity,Preds),
occ(els(Pred,Arity,Cls),Relations),
occ(Cl,Cls).

/* pick out the function symbol and arity of an equation */
defining_fun(Vars-(LHS=RHS:-Body),Fun,Arity):-
\[functor(LHS,Fun,Arity).
defining_fun(Vars-(LHS=RHS),Fun,Arity):-
\[functor(LHS,Fun,Arity).

/* pick out the predicate and arity of the clause */
defining_pred(Vars-(Head:-Body),Pred,Arity):-
\[functor(Head,Pred,Arity).
defining_pred(Vars-Head,Pred,Arity):-
\[functor(Head,Pred,Arity).
%
% map the relations within the class body into Prolog-style clauses
Appendix B: Class template preprocessor

\[
\text{tr\_methods([], Ax, Ax, Rls, Rls, Defnd, LbL, Super):-!}.
\]
\[
\text{tr\_methods([cls(Pred,Arity,Clauses)|Reis], Ax, Az, [P(Cls)|R],}
\]
\[
\text{Rs,Defnd,LbL,Super):-}
\]
\[
\text{termin(Clauses),}
\]
\[
\text{local\_pred(LbL,Pred,P),}
\]
\[
\text{tr\_rel(Clauses, Cls, Cli, Ax, Ay, Defnd, LbL),!},
\]
\[
\text{super\_clause(Super, Cls, [], Pred, Arity, LbL), % construct}
\]
\[
\text{super reference}
\]
\[
\text{tr\_methods(Reis, Ay, Az, R, Rs, Defnd, LbL, Super).}
\]
\[
\%
\%
\%
\]
\[
\text{map a relation into the appropriate suite of clauses}
\]
\[
\%
\%
\%
\]
\[
\text{tr\_rel([], C, C, Aux, Aux, Defnd, LbL).}
\]
\[
\text{tr\_rel([Cl|Cls], [P(CL|PC), PCls, Aux, Ax, Defnd, LbL]):-}
\]
\[
\text{tr\_clause(Cl, PCl, Aux, Ax1, Defnd, LbL),}
\]
\[
\text{tr\_rel(Cls, PC, PCls, Ax1, Ax, Defnd, LbL).}
\]
\[
\%
\%
\%
\]
\[
\text{map the functions within the class body into Prolog style clauses}
\]
\[
\%
\%
\%
\]
\[
\text{tr\_functions([], Ax, Ax, Rls, Rls, Defnd, LbL):-!}.
\]
\[
\text{tr\_functions([eqs(Fun,Arity,Eqs)|Fs], Ax, Az, [P(Cls)|R], Rs,}
\]
\[
\text{Defnd, LbL):-}
\]
\[
\text{termin(Eqs),}
\]
\[
\text{local\_fun(LbL,Fun,P),}
\]
\[
\text{tr\_fun(Eqs, Cls, [], Ax, Ay, Defnd, LbL),}
\]
\[
\text{tr\_functions(Fs, Ay, Az, R, Rs, Defnd, LbL).}
\]
\[
\text{tr\_fun([], C, C, Aux, Aux, Defnd, LbL).}
\]
\[
\text{tr\_fun([Eq|Eqs], [P(Eq|PC), PEqs, Ax, Az, Defnd, LbL):-}
\]
\[
\text{tr\_eqn(Eq, PEq, Ax, Ay, Defnd, LbL),}
\]
\[
\text{tr\_fun(Eqs, PC, PEqs, Ay, Az, Defnd, LbL).}
\]
\[
\%
\%
\%
\]
\[
\text{the clauses that represent the predicate entry points into the}
\]
\[
\%
\%
\%
\]
\[
\text{mk\_entry\_rels([],Lbs,OCls,OCls).}
\]
\[
\text{mk\_entry\_rels([Pr|Ar|Preds],Lbs,[['\_?lb\:', '\_?sf\:'|V]-}
\]
\[
\text{(Lbs(Atom,'\_?lb\:', '\_?sf\:'):-!,B)|OCls], OCls1):-}
\]
\[
\text{functor(At,Pr,Ar),}
\]
\[
\text{toground(At,Atom,V),}
\]
local_pred(Lbs, Pr, LocP),
append(V, ['_?lb?', '_?sf?'], LocA),
B=..[LocP|LocA],
mk_entry_rels(Preds, Lbs, OCls, OClsl).

% the clauses that represent the function entry points
% mk_entry_funs([], Lbs, OCls, OClsl).
mk_entry_funs([F/Ar|Funs], Lbs,
[['_?lb?','_?sf?']V|Lbs(Atom,'_?lb?','JPsf?'):=-
!, B) [OCls, OClsl]):-
Arl is Ar+1,
concat(F, '*', Fn),
functor(At, Fn, Arl),
toground(At, Atom, V),
local_fun(Lbs, F, LocP),
append(V, ['_?lb?','_?sf?'], LocA),
B=..[LocP|LocA],
mk_entry_funs(Funs, Lbs, OCls, OClsl).

% final entry point which picks up on the class rules
% mk_super_rule(_, Cls, Cls, absent):!.
mk_super_rule(LbS, [['_?a?','_?1?1,'_?s?']-
Lbs('_?a? * ,  _?1?1,  _?s?') :-SP('_?a? 1,  '_?1?1,
*_?s?1))|Cls], Cls, present):-
concat(LbS, ':super', SP).

/* convert a class body axiom into the set of clauses needed to implement it */
tr_clause(V-(H:-B), [['_?lb?','_?sf?']|Vars]|(Head:-Body), Aux, Ax,
Defnd, Lbs(LbL)):-!,
H=..[HPr|Hargs],
functor(H, _, HAr),
tr_list(Hargs, Nargs, [], SGH, V, Aux, Axl, 0(V), Vint,
Defnd, Lbs(LbL)),
form_local_pred(Lbs, HPr, HAr, Nargs, Head, Defnd),
tr_body(B, Bdy, V, Axl, Ax, Vint, (Vars), Defnd, Lbs(LbL)),
sim_goals((Bdy, SGH), Body).
tr_clause(V-H, [['_?lb?','_?sf?']|Vars]|(Head:-Body), Aux, Ax, Defnd,
Lbs(LbL)):-!
H=..[HPr|Hargs],
functor(H,_,HAr),
tr_list(Hargs, Nargs, [], SGH, V, Aux, Ax, 0(V), _(_(Vars)),
       Defnd, LbS(LbL)),
form_local_pred(LbS, HPr, HAr, Nargs, Head, Defnd),
sim_goals(SGH, Body).

/* compile an equation - similar to compiling a normal clause */
tr_eqn(V-(LHS=RHS:-B),['_?lb?','_?sf?'|Vars]-(Head:-Body),Aux, Ax,
       Defnd, LbL):-!,
LHS=..[F|A], % split LHS into function and arguments
append(A,[RHS],Args),
functor(LHS,_,Ar),
tr_list(Args, Nargs, [], SGH, V, Aux, Ax1, 0(V), Vint,
       Defnd,LbL),
form_fun_goal(F, Ar, Nargs, V, LbL, Head, Defnd),
tr_body(B, Bdy, V, Ax1, Ax, Vint, _(_(Vars)), Defnd, LbL),
sim_goals((Bdy,! , SGH), Body).
tr_eqn(V-(LHS=RHS), Clause, Aux, Ax, Defnd, LbL):-
tr_eqn(V-(LHS=RHS:-true), Clause, Aux, Ax, Defnd, LbL).

/* Convert conditions into their local form, leaving labelled
conditions
in a special form
Also extract any disjunctions, negations and arrows from a clause,
leaving just a conjunction of calls to deal with.
Also extract any functional expressions and convert them into
calls to predicates */

/* tr_body examines each of the call in a body in turn, and
constructs auxilliary programs as necessary */
tr_body(true, true, Vars, Aux, Aux, Vno, Vno, Defnd, LbL).
tr_body((At,B), (Na,Nb), Vars, Aux, Oux, Vno, Vout, Defnd, LbL):-!
tr_body(At, Na, Vars, Aux, X1, Vno, Vi, Defnd, LbL),
tr_body(B, Nb, Vars, X1, Oux, Vi, Vout, Defnd, LbL).

/* label movement rules */
tr_body(Label:(E,O),Auxill,Vars,Aux,Oux, Vno, Vout, Defnd, LbL):-
tr_body((Label:E,Label:O),Auxill,Vars,Aux,Oux,Vno,Vout, Defnd,
Appendix B: Class template preprocessor

tr_body(Label:(E;0), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL):-
    tr_body((Label:E;Label:0), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL).

tr_body(Label:(\+E), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL):-
    tr_body(\+(Label:E), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL).

tr_body(Label:(not E), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL):-
    tr_body(\+(Label:E), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL).

tr_body(L:(E->Th;El), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL):-
    tr_body((L:E->L:Th;L:El), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL).

tr_body(Label:(Lab:C), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL):-
    tr_body((Lab:C), Auxill, Vars, Aux, Oux, Vno, Vout, Defnd, LbL).

% Unwind an individual call, looking for ; -> not etc.
% conditional

% tr_body((T->Th), (Test->Then), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
%    tr_body(T, Test, V, Aux, Ax, Vno, Vout, Defnd, LbL),
%    tr_body(Th, Then, V, Ax, Ax, V, Vout, Defnd, LbL).

% disjunction
tr_body((Ei;Or), (Either; Orr), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
    tr_body(Ei, Either, V, Aux, Ax, Vno, Vl, Defnd, LbL),
    tr_body(Or, Orr, V, Ax, Ax, Vl, Vout, Defnd, LbL).

% negation
tr_body(\+NT, \+Not, V, Aux, Ax, Vno, Vout, Defnd, LbL):-
    tr_body(NT, Not, V, Aux, Ax, Vno, Vout, Defnd, LbL).

% a special case where a condition is an equality in the body
tr_body((LHS=RHS), G, V, Aux, Ax, Vno, Vout, Defnd, LbL):-
    \+on(LHS,V) ->
    tr_body_eq(LHS, RHS, G, V, Aux, Ax, Vno, Vout, Defnd, LbL);
    \+on(RHS,V) ->
    tr_body_eq(RHS, LHS, G, V, Aux, Ax, Vno, Vout, Defnd, LbL);
    G=(LHS=RHS), Vout=Vno, Ax=Aux.
% super labelled call
tr_body(super:Call, Goal, Vars, Aux, Ax, Vno, Vout, Defnd,
   LbS(LbL)):-!,
   Call=.>[CPr|Cargs],
   tr_list(Cargs, Nargs, [], SGC, Vars, Aux, Ax, Vno, Vout, Defnd,
   LbS(LbL)),
   Cll=.>[CPr|Nargs],
   local_pred(LbS, 'super', LocP),
   sim_goals((SGC, LocP(Cll, '_?lb?', '_?sf?')), Goal).

% labelled condition - generate the special labelled call.
tr_body(Label:Call, Goal, Vars, Aux, Ax, Vno, Vout, Defnd, LbL):-
   tr_term(Label, Lb, SGL, Vars, Aux, Ax1, Vno, Vi(LVi), Defnd,
   LbL),
   Call=.>[CPr|Callargs],
   tr_list(Callargs,Nargs, [], SGC, Vars, Ax1, Ax, Vi(LVi), Vout,
   Defnd, LbL),
   Cll=>[CPr|Nargs],
   ((on(Lb,LVi);on(CPr,LVi))->
      sim_goals((SGL, SGC, '??:?'(Lb,Cll,Lb)), Goal);
      functor(Lb,LbSymb,_,)
      sim_goals((SGL,SGC,LbSymb(Cll,Lb,Lb)),Goal)).

% cuts are left alone
tr_body(!, !, Vars, Aux, Aux, Vno, Vno, Defnd, LbL):-!.

% basic condition - extract any expressions from inside it
tr_body(Call, Goal, Vars, Aux, Ax, Vno, Vi(LVi), Defnd,
   LbS(LbL)):-
   Call=.>[CPr|Cargs],
   tr_list(Cargs, Nargs, [], SGC, Vars, Aux, Ax, Vno, Vi(LVi),
   Defnd, LbS(LbL)),
   (on(CPr,LVi)->sim_goals((SGC,'?apply?'(CPr, Nargs, '_?lb?',
      '_?sf?')), Goal);
      functor(Call,_|CAR),
      form_local_pred(LbS, CPr, CAR, Nargs, LocPred, Defnd),
      sim_goals((SGC,LocPred),Goal)).

form_local_pred(Lb, $(Pr), Ar, Args, Atom, Defnd):-
   Atom=>[Pr|Args],!.
form_local_pred(Lb, Pr, Ar, Args, Atom, (Funs,Preds)):-
Appendix B: Class template preprocessor

on(Pr/Ar,Preds)->
    (append(Args,['/_?lb?','/_?sf?'],AArgs),
     local_pred(Lb,Pr,LocP), % construct local predicate
     Atom=..[LocP|AArgs]);

A=..[Pr|Args],
Atom=lb(A,'/_?lb?','/_?sf?').

tr_body_eq(LHS, RHS, Goal, V, Aux, Vno, Vout, Defnd, LbL):-
    LHS=..[F|A], % split LHS into function and arguments
    append(A,[RHS],Args),
    functor(LHS,_,Ar),
    tr_list(Args, Nargs, [], SG, V, Aux, Ax, Vno, Vout, Defnd,
           LbL),
    form_fun_goal(F, Ar, Nargs, V, LbL, GL, Defnd),
    sim_goals((SG,GL),Goal).

/*
   translate the class rules of a complete class template
   program. Two sets of clauses are generated - those for
   normal class rules and those for overriding class rules
*/

tr_rules([],_Rels, _Rels, absent, absent, _):-!.
tr_rules(Rules, [LbSuper(Cls)|Rels], Rlo, Inherit, present, Lb):-
    concat(Lb,':super',LbSuper),
    tr_all_rules(Rules, Cls, Lb),
    tr_over_rules(Rules, Over, [], absent, Inherit),
    (Inherit=present -> concat(Lb,':inherit',LblInherit),
     Rels = [LblInherit(Over)|Rlo];
     Rels=Rlo).

tr_all_rules([],[],_,_).
tr_all_rules([V-_(Lbl,Mbl)|Rules], % handles both <= and <=
             [['/_?at?','/_?sf?']|V]-_(Lbs('/_?at?',Lbl,'/_?sf?'))-_(Mbs('/_?at?',Mbl,'/_?sf?'))|Clss],Lb):-
    functor(Mbl,Mbs,_,),
    concat(Lb,':super',Lbs),
    tr_all_rules(Rules, Cls, Lb).

tr_over_rules([],Cls,Cls,A,A).
tr_over_rules([V-_(Lbl<=Mbl)|Rules], % handles only <=
              [['/_?at?','/_?sf?']|V]-_(Lbo('/_?at?',Lbl,'/_?sf?'))-_(Mbs('/_?at?','/_?sf?'))|Clss],OC,_,present):-
    functor(Lbl,Lbs,_,),
Appendix B: Class template preprocessor

functor(Mbl,MbS,_) ,
concat(LbS,':inherit',LbO),
tr_over_rules(Rules,Cls,OC,_,)._.

tr_over_rules([V-Lbl<<MbI] | Rules], Cls,OC,A,B):- % ignore <<
  tr_over_rules(Rules,Cls,OC,A,B).

super_clause(absent, Cls, Cls, Pred, Arity, LbL):-!.
super_clause(present, [['_?lb?','_?sf?']|V]-
  (Hd:-LP(Atom,'_?lb?','_?sf?'))[Cls], Cls, Pred, Arity,
  LbS(_)):=-
  concat(LbS, ': inherit', LP),
  local_pred(LbS, Pred, LocP),
  length(As, Arity), % form the basic arguments
toground(As, Args, V),
  Atom=..[Pred|Args],
  append(Args,['_?lb?','_?sf?'], HA),
  Hd=..[LocP|HA],!.

/* tr_term translates a term which may be a functional expression
into a regular term, together with a conjunction of goals, and a
set of auxiliary clause.

If the term is a lambda expression or set abstraction then the
lambda clause is computed and returned in the auxiliary list of
clauses */

% quoted expression - no interpretation
tr_term('X, Y, G, V, Aux, Ax, Vno, Vout, Defnd, LbL):-!,
  tr_quoted(X, Y, G, V, Aux, Ax, Vno, Vout, Defnd, LbL).

% hashed expression - ignore at this level
tr_term(#X, Y, SG, V, Aux, Ax, Vno, Vout, Defnd, LbL):-!,
  tr_term(X, Y, SG, V, Aux, Ax, Vno, Vout, Defnd, LbL).

% lambda expression - generate an auxiliary lambda clause
tr_term(Lambda«Expr,T, true, V, Aux, Ax, Vno,Vno,Defnd, LbL):-!,
  Lambda=..[lambda|Largs],!, % get the parameters of the lambda
  tr_lambda(Largs, Expr, T, V, Aux, Ax, Defnd, LbL).

% similar case with a set abstraction
tr_term((Tuple|Condition),T, true, V, Aux, Ax, Vno,Vno,Defnd,
Appendix B: Class template preprocessor

LblL):-!
   tr_union({Tuple|Condition}, T, V, Aux, Ax, Defnd, LbL).

% union set abstraction
tr_term(Either'|'|'Or, T, true, V, Aux, Ax, Vno,Vno,Defnd, LbL):-!
   tr_union(Either'|'|'Or, T, V, Aux, Ax, Defnd, LbL).

% declare a constant as a function symbol.
tr_term(F*, Fn, (FG,G), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
   tr_term(F, F0, FG, V, Aux, Ax, Vno, N(Vi), Defnd, LbL),
   tr_fun_symbol(F0, Fn, G, Vi, N(Vi), Vout, LbL).

% external class template function call
tr_term(Class: Var, T, G, V, Aux, Ax, Vno, Vout, Defnd, LbL):-
   on(Var, V),!
   new_var_name(Vno, Vn, T),
   tr_body(Class:'?#?'(Var,T), G, V, Aux, Ax, Vn, Vout, Defnd, LbL).

tr_term(Class:Term, T, G, V, Aux, Ax, Vno, Vout, Defnd, LbL):-!
   new_var_name(Vno, Vn, T),
   Term=..[Fun|A],
   append(A,[T],B),
   concat(Fun, '*', Fn),
   Goal=..[Fn|B],
   tr_body(Class:Goal, G, V, Aux, Ax, Vn, Vout, Defnd, LbL).

% self keyword reference
tr_term(self, '_?sf?', true, V, Aux, Aux, Vn, Vn, Defnd, LbL):-!.

% a list pair is examined internally
tr_term([H|T], [HS|HT], (GH,GT), V, Aux, Ax, Vno, Vout, Defnd, LbL):-!
   tr_term(H, HS, GH, V, Aux, Ax1, Vno, Vi, Defnd, LbL),
   tr_term(T, HT, GT, V, Ax1, Ax, Vi, Vout, Defnd, LbL).

% number term - no interpretation
tr_term(X, X, true, V, Aux, Aux, Vout, Defnd, LbL):-!
   number(X),!.

% variable term - no interpretation
tr_term(X, X, true, V, Aux, Aux, Vout, Defnd, LbL):-
atom(X),
on(X,V),!.

% a variable which is a reference to a label argument
tr_term(X,X, arg(N,'(?lb?'),X), V, Aux, Aux, L(F), L([X|F]), Defnd, L(LArgs)):-
atom(X),
indexed_occ(X,LArgs,N),!.

% if a term is atomic - it may be a constant function
tr_term(T, S, G, V, Aux, Vno, Vout, (Funs,Preds), LbL):-
atom(T),!
(occ(T/0,Funs)->
    new_var_name(Vno,Vout,S),  G=T(S);
    S=T, G=true, Vno=Vout).

% a complex term is ripped apart and rebuilt in a new guise
tr_term(T, S, G f, V, Aux, Ax, Vno, Vout, Defnd, LbL):=
    functor(T,F,Ar),
    T=.[F|A],
    tr_complex(F, Ar, A, S, G, V, Aux, Ax, Vno, Vout, Defnd, LbL).

% examine a complex term
% a reducible expression ...
tr_complex($F$, Ar, A, S, (FG,SG,G), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
    tr_function(F, Ar, Fn, FG, V, Aux, Ax1, Vno, VN(Vi), Defnd, LbL),!
    new_var_name(VN(Vi), Vn, S),
    tr_list(A, B, [S], SG, V, Axl, Ax, Vn, Vout, Defnd, LbL),
    form_fun_goal(Fn, Ar, B, Vi, LbL, G, Defnd).
tr_complex(F, Ar, A, S, (FG,SG,G), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
    tr_function(F, Ar, Fn, FG, V, Aux, Ax1, Vno, VN(Vi), Defnd, LbL),!
    new_var_name(VN(Vi), Vn, S),
    tr_list(A, B, [S], SG, V, Axl, Ax, Vn, Vout, Defnd, LbL),
    form_fun_goal(Fn, Ar, B, Vi, LbL, G, Defnd).
% a normal free function ...
tr_complex(F, Ar, A, S, GL, V, Aux, Ax, Vno, Vout, Defnd, LbL):-
    atom(F),!,
    tr_list(A, B, [], GL, V, Aux, Ax, Vno, Vout, Defnd, LbL),
Appendix B: Gass template preprocessor

% handle a function symbol
tr_function(F, Ar, F, true, V, Aux, Vno, Vno, (Funs,Preds), LbL):-
  atom(F),!,
  (on(F,V);on(F/Ar,Funs)).
tr_function(F, Ar, S, true, V, Aux, Ax, Vno, Vout, Defnd, LbL):-
  F=..[FC|FA],
  functor(F,FC,FAr),
  tr_complex(FC, FAr, FA, S, V, Aux, Ax, Vno, Vout, Defnd, LbL).

% has the user has declared a symbol to be the name of a function?
tr_fun_symbol(F, S, putchar('?','?lb?','_?sf?',S), V, Vno, Vout, LbL(Largs)):=
  (on(F,V);on(F,Largs)),!,
  new_var_name(Vno, Vout, S).
tr_fun_symbol(F, F n ('?lb?','_?sf?'), true, V, Vno, Vno, LbL):=
  atom(F),!
  concat(F,'**',Fn).
tr_fun_symbol(F, F, true, V, Vno, Vno, LbL).

% translate a list of terms...
tr_list([], L, L, true, V, Aux, Vno, Vno, Defnd, LbL).
tr_list([T|L], [S|K], R, (G,GL), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
  tr_term(T, S, G, V, Aux, Ax, Vn, Vout, Defnd, LbL),

% tr_quoted inspects a quoted term looking for the unquote symbol
# tr_quoted(#X, Y, G, V, Aux, Ax, Vno, Vout, Defnd, LbL):-!,
  tr_term(X, Y, G, V, Aux, Ax, Vno, Vout, Defnd, LbL).
% unquote part of term
% a variable which is a reference to a label argument
tr_quoted(X, X, arg(N,'?lb?','?sf?'), X, V, Aux, Aux, L(F), L([X|F]),
  Defnd, _(LArgs)):=
  atom(X),
  indexed_occ(X,LArgs,N),!.
tr_quoted(X, X, true, V, Aux, Aux, Vno, Vno, Defnd, LbL):-
  atomic(X),!.
tr_quoted(X, Y, (FG,AG), V, Aux, Ax, Vno, Vout, Defnd, LbL):-
Appendix B: Class template preprocessor

\[
x = \ldots [F|A], \\
tr\_quoted(F, FY, FG, V, Aux, Axl, Vno, V1, Defnd, LbL), \\
tr\_quoted\_list(A, AY, AG, V, Axl, Ax, V1, Vout, Defnd, LbL), \\
Y = \ldots [FY|AY].
\]

\[
tr\_quoted\_list([], [], true, V, Aux, Aux, Vno, Vno, Defnd, LbL).
\]

\[
tr\_quoted\_list([T|L], [S|K], (G,GL), V, Aux, Ax, Vno, Vout, Defnd, LbL):- \\
tr\_quoted(T, S, G, V, Aux, Axl, Vno, Vn, Defnd, LbL), \\
tr\_quoted\_list(L, K, GL, V, Axl, Ax, Vn, Vout, Defnd, LbL).
\]

\% form a sub-goal that will constrain the value of a variable
% form\_fun\_goal(F,Ar,B,V,LbS(LbL),'?apply?'(F,B,'?lb?'','?sf?''), \\
Defnd):- \\
on(F,V),!.
form\_fun\_goal($(F), Ar, Args, V, LbL, Goal, Defnd):- \\
Goal=..[F|Args],!.
form\_fun\_goal(F, Ar, Args, V, Lb(LbL), Goal, (Funs,Preds)):- \\
on(F/Ar,Funs):- \\
\ldots (append(Args,'?lb?', '?sf?' ),AArgs), \\
local\_fun(Lb,F,LocP), \\
Goal=..[LocP|AArgs]); \\
concat(F,'*1,Fn), \\
A=..[Fn|Args], \\
Goal=Lb(A,'?lb?', '?sf?').
\%
% simplify the complex nesting of sub-goals (mostly trues of course)
%
sim\_goals(G, S):- \\
sim\_left(G, L), \\
sim\_right(L, S).

sim\_left((true,A), L):- \\
sim\_left(A, L).

sim\_left(((A,B),C), L):- \\
sim\_left((A,B,C), L).

sim\_left((A,B), (SA;SB)):- \\
sim\_goals(A,SA),
sim_goals(B,SB).
sim_left((A->B), (SA->SB)):-
sim_goals(A,SA),
sim_goals(B,SB).
sim_left(\+A, \+SA):-
sim_goals(A,SA).
sim_left(A,A).

sim_right((A,B), C):-
sim_goals(B, SB),
(SB=true->A=C;C=(A,SB)).
sim_right(A,A).

% translate lambda expressions and set abstractions into
auxilliary clauses
%
tr_lambda(Largs, Expr, Form, V,
    Ix([LV-(LS(Hed,'_?lb?','_?sf?'):-Body)|R]),AxS,Defnd,LbL):-
  free_vars(V, Expr, Largs, Free, LcV),
  new_label(Ix,Iy,LSymb,LbL),
  tr_term(Expr, E, SGI, LcV, Iy(R), Axl,
      0(['_?lb?','_?sf?'|LcV]), Vi, Defnd, LbL),
  sim_goals((SGO,SGI), Body),% the generated body of the lambda
  LM=..[LSymb|Free],% the 'predicate' has free vars
  Hed =.. [LM|LA],% the predication itself
  functor(LbL,LS,_)..
%
% determine the free and local variables in a lambda expression
%
free_vars([], Ex, La, [],[]):-!.
free_vars([V|Vars], Expr, Largs, Free, [V|Locals]):-
  arb_mem(V, Largs),!,
  free_vars(Vars, Expr, Largs, Free, Locals).
free_vars([V|Vars], Expr, Largs, [V|Free], [V|Locals]):-
  arb_mem(V, Expr),!,
  free_vars(Vars, Expr, Largs, Free, Locals).
free_vars([V|Vars], Expr, Largs, Free, Locals):-
  free_vars(Vars, Expr, Largs, Free, Locals).
% tr_union handle set abstractions and unions.

tr_union(Union, Form, V, Ix(ACls), Ax, Defnd, LbL):-
    union_free_vars(V, Union, Free),  % extract all the free vars
    new_label(Ix, Iy, LSymb, LbL),  % construct a new symbol
    Form =.. [LSymb, '_?lb?', '_?sf?' |Free],
    Pred =.. [LSymb | Free],
    tr_union_arms(Union, LSymb, Pred, V, ACls, R, Iy(R), Ax, Defnd,
                  LbL).

tr_union_arms(Either1 | 'Or, LS, Form, V, [ACls], ACls, Aux, Ax,
               Defnd, LbL):-!
    tr_arm(Either, LS, Form, V, A, Aux, Ax1, Defnd, LbL),
    tr_union_arms(Or, LS, Form, V, Cls, ACls, Ax1, Ax, Defnd, LbL).

tr_union_arms(Set, LS, Form, V, [A|Cls], CLs, Aux, Ax, Defnd,
               LbL):-
    tr_arm(Set, LS, Form, V, LV-(LS(Hed, '_?lb?','_?sf?'):-Body),
           Aux, Ax, Defnd, LbL):-!,
    local_vars(V, Set, LcV),  % pick up the local variables
    (Set=(Tpl/Exl|Cnd)->true;Set=(Tpl|Cnd)),
    tr_term(Tpl, TTpl, SGI, LcV, Aux, Ax1,
            0(['_?lb?', '_?sf?'|LcV]), Vint, Defnd, LbL),
    tr_body(Cnd, LB, LcV, Ax1, Ax, Vint, _LV, Defnd, LbL),
    sim_goals((LB, SGI), Body),
    comma_list(TTpl, L),  % the head of the lambda body clause
    Hed =.. [Form|L],
    functor(LbL, LS, _).

% local_vars extract from the whole list of variables the local ones
%
local_vars([], Ex, []):-!.
local_vars([V|Vars], Expr, [V|Locals]):-
    arb_mem(V, Expr),!,
    local_vars(Vars, Expr, Locals).
local_vars([V|Vars], Expr, Locals):-
    local_vars(Vars, Expr, Locals).

% determine the free variables in a set union
Appendix B: Class template preprocessor

union_free_vars([], Union, []):-!.
union_free_vars([V|Vars], Union, [V|Free]):-
    union_free_var(Union, V),!,
    union_free_vars(Vars, Union, Free).
union_free_vars([V|Vars], Union, Free):-
    union_free_vars(Vars, Union, Free).

union_free_var({Tpl|Cnd}, V) :-
    \+ arb_mem(V, Tpl),
    arb_mem(V, Cnd),!.
union_free_var(Either'||'Or, V) :-
    union_free_var(Either, V),!.
union_free_var(Either'||'Or, V) :-
    union_free_var(Or, V).

comma_list((T1,R), [T1|L1]) :-!,
    comma_list(R, LI).
comma_list(T, [T])  .

/* library functions used by the compiled code */

'??:?'(Label, Atom, Self):-
    functor(Atom, P, _),
    atom(P),!,
    functor(Label, L, _),
    L(Atom, Label, Self).
'??:?'(_, A, _):-
    A =..[P|Args],
    P =.. [Pr, Label, Self|F],
    NP =.. [Pr|F],
    Atom =.. [NP |Args],
    functor(Label, Lb, _),
    Lb(Atom, Label, Self).

'?apply?'(P, Args, Label, Self):-
    atom(P),!,
    functor(Label, L, _),
    Atom =.. [P|Args],
    L(Atom, Label, Self).
'?apply?'(P, Args, _, _):-
    P =.. [Pr, Label, Self|F],
Appendix B: Class template preprocessor

\[
\begin{align*}
NP &= [Pr|F], \\
Atom &= [NP | Args], \\
functor(Label, Lb, _), \\
Lb(Atom, Label, Self).
\end{align*}
\]

\[L : C (L, C, L).\]

\[\text{'?*?'(Symbol, Label, Self, Symbol):}- \\
\text{compound(Symbol),}!, \\
'\text{?*?'}(Symbol, Label, Self, F(Label, Self)):- \\
\text{atom(Symbol),} \\
\text{concat(Symbol, '*', F)}.\]

\[\text{system('?#?')(Exp, Result), Label, Self):-!,} \\
\text{Exp=}..[F|A], \\
\text{append(A, [Result], B),} \\
'\text{?apply?'}(F, B, Label, Self).\]

\[\text{system(Atom, _, _):-} \]
\text{Atom.}\% 

\% define some miscellaneous primitives for use by the 
\% preprocessor 
\%
\[\text{tuple(T)}:-'tpl%i'(T),!.
\]
\[\text{tuple(T, L)}:-'tpl%i'(T), 'tpl%i'(T, L).
\]

\[\text{nth(T, N, E)} :- 'tpl%i'(T), 'nth%i'(T, N, D), D=E.\]

\[\text{/* occ is unsafe list membership */} \]
\[\text{occ(Var, [Var|_]):-!}.
\]
\[\text{occ(Var, [_|List]):-} \]
\[\text{occ(Var, List).} \]

\[\text{/* no_occ is non-membership */} \]
\[\text{no_occ(_[..]):-!}.
\]
\[\text{no_occ(E, [T|L]):-} \]
\[\text{E\=T,} \]
\[\text{no_occ(E, L).} \]

\[\text{/* indexed_occ looks an element up in a list and returns} \]
\text{its position */} \]
\[\text{indexed_occ(X, L, N)} :- \text{ix_o(X, L, 1, N)}.\]
ix_o(X,[X|L],N,N):-!.
ix_o(X,[_|L],P,N):-
  ++(P,1,P1),
  ix_o(X,L,P1,N).

/* arb-member member of an arbitrary structure */
arb_member(El, Term):-
  arb_mem(El, Term),
  !.
/* this is needed to avoid a call to negation and hence
meta_variable */
not_arb_member(El, Term):-
  arb_mem(El, Term),
  !,
  false.
not_arb_member(El, Term).
arb_mem(El, El).
arb_mem(El, [E| List])-:
arb_mem(El, E).
arb_mem(El, [E| List]):-
arb_mem(El, List).
arb_mem(El, Tuple):-
tuple(Tuple, Length),
0<Length,
arb_tuple_mem(El, 1, Length, Tuple).
arb_tuple_mem(El, Pos, Length, Tuple):-
nth(Tuple, Pos, E),
arb_mem(El, E).
arb_tuple_mem(El, Pos, Length, Tuple):-
  Pos<Length,
  ++(Pos, 1, Ps),
arb_tuple_mem(El, Ps, Length, Tuple).

% % termin is used to complete a list
% termin([]):-!.
termin([|L]):=termin(L).

% new_var_name constructs a new variable symbol of the form
Appendix B: Class template preprocessor

new_var_name(Vno(List), Vn([V|List]), V):-
    concat('?', Vno, V),
    Vn is Vno+1.

new_label(Lbn, Lbl, Lb, Prefix):-
    local_pred(Prefix, Lbn, Lb),
    Lbl is Lbn+1.

local_pred(Prefix(_), Pred, Symbol):-!
    concat(Prefix, ':', I),
    concat(I, Pred, Symbol).

local_pred(Prefix, Pred, Symbol):-
    concat(Prefix, ':', I),
    concat(I, Pred, Symbol).

local_fun(Prefix(_), Fun, Symbol):-!
    concat(Prefix, ':', I),
    concat(I, Fun, II),
    concat(II, '*', Symbol).

local_fun(Prefix, Fun, Symbol):-
    concat(Prefix, ':', I),
    concat(I, Fun, II),
    concat(II, '*', Symbol).

disp_prolog(Reis):-
    get_prop('display', '_display', '_display', '_display'),
    disp_rels(Reis).

disp_prolog(_).
disp_rels([]).
disp_rels([Rel(Cls)|R]):-
    writeqseqnl(user,[Rel,:
    disp_clauses(Cls),
disp_rels(R).
% display clauses which have been generated by the preprocessor
% display_clauses(Cls):-
disp_clauses(Cls),!.

disp_clauses([]).
disp_clauses([V-Cl|Cls]):-
display_cl(Cl,V),
disp_clauses(Cls).

display_cl((Head:-true),V):-!,
    writeseq(user,[V,'— M ']),
    writeq(user,Head,V),
    writeln(user,'.').
display_cl((Head:-Body),V):-!,
... writeseq(user,[V,'— M ']),
... writeq(user,Head,V),
display_body(Body,V,'— M ').
display_cl(Head,V):-
    writeseq(user,[V,'— M ']),
    writeq(user,Head,V),
    writeln(user,'.').

display_body((A,B),V,Con):-!,
    write(user,Con),
    writeq(user,A,V),
display_body(B,V,'— M ').
display_body(A,V,Con):-
    write(user,Con),
    writeq(user,A,V),
    write(user,'— M ').
Appendix C: Example class template programs

lists:
    append([],X,X).
    append([E|X],Y,[E|Z]):-
        append(X,Y,Z).
    naive([],[]).
    naive([E|L],R):-
        naive(L,I),
        append(I,[E],R).
    merge([],L)=L.
    merge(L,[])=L.
    merge([D|L1],[E|L2])=[D|merge(L1,[E|L2])] :- D<E.
    merge([D|L1],[E|L2])=[E|merge([D|L1],L2)] :- E=<D.

max(X,Y)=Z :- X>Y -> Z=X ; Z=Y
}

simple(Order):{
    sort([],[]).
    sort([X|L])= insert(X,sort(L)).

    insert(X,[])=[X].
    insert(X,[D|L])=[X,D|L] Order:less(X,D).
    insert(X,[D|L])=[D|insert(X,L)] :- \+Order:less(X,D)
}

natural:{
    less(X,Y):- X<Y
}

natural<<system.

descend:{
    less(X,Y):- X>Y
}

Appendix C: Example class template programs

descend<<system.

cart(01,02):{-
  less((X,Y),(U,V)):-01:less(X,U).
  less((X,Y),(X,V)):-02:less(Y,V)
}.
goal:{
  apply(F,X,F(X)).
  test1(X,simple(descend):sort(X)).
  test2(X,simple(cart(natural,descend)):sort(X))
}.
goal<<system.

person(Ag):{-
  age(Ag).
  likes(X):-X:age(A),A<10.
  likes(X):-X:like(self)
}.
person(Ag)<=life_form.
person(Ag)<<system.
Appendix C: Example class template programs

The simple and associated class templates above are compiled into the following sets of clauses:

cart :
[_?lb?, _?sf?, _1, _2] -
cart(less(_1, _2), _?lb?, _?sf?):-
!,
'cart:less'(_1, _2, _?lb?, _?sf?).
'cart:less' :
[_?lb?, _?sf?, 01, X, Y, U, V] -
'cart:less'((X,Y), (U,V), _?lb?, _?sf?):-
arg(1, _?lb?, 01),
?:?(01, less(X, U), 01).
[_?lb?, _?sf?, 02, X, Y, V] -
'cart:less'((X,Y), (X,V), _?lb?, _?sf?):-
arg(2, _?lb?, 02),
?:?(02, less(Y, V), 02).
descend :
[_?lb?, _?sf?, _1, _2] -
descend(less(_1, _2), _?lb?, _?sf?):-
!,
'descend:less'(_1, _2, _?lb?, _?sf?).
[_?a?, _?1?, _?s?] -
descend(_?a?, _?1?, _?s?):-
'descend:super'(_?a?, _?1?, _?s?).
'descend:less' :
[_?lb?, _?sf?, X, Y] -
'descend:less'(X, Y, _?lb?, _?sf?):-
descend(X>Y, _?lb?, _?sf?).
'descend:super' :
[_?at?, _?sf?] -
'descend:super'(_?at?, descend, _?sf?):-
system(_?at?, system, _?sf?).
goal :
[_?lb?, _?sf?, _1, _2, _3] -
goal(apply(_1, _2, _3), _?lb?, _?sf?):-
!,
'goal:apply'(_1, _2, _3, _?lb?, _?sf?).
Appendix C: Example class template programs

[?lb?, ?sf?, _1, _2] -
  goal(test1(_, _2), ?lb?, ?sf?):-
    !,
    'goal: test1'(_, _2, ?lb?, ?sf?).

[?lb?, ?sf?, _1, _2] -
  goal(test2(_, _2), ?lb?, ?sf?):-
    !,
    'goal: test2'(_, _2, ?lb?, ?sf?).

[?a?, _?1?, _?s?] -
  goal(_?a?, _?1?, _?s?):-
    'goal: super'(_?a?, _?1?, _?s?).

'goal: apply' :
[?lb?, ?sf?, _0, F, X] -
  'goal: apply'(F, X, _?0, ?lb?, ?sf?):-
    '?apply'?(F, [X, _?0], ?lb?, ?sf?).

'goal: test1' :
[?lb?, ?sf?, _0, X] -
  goal(test1(X, _?0, ?lb?, ?sf?):-
    .
    .
    simple('sort*'(_, _?0), simple(descend),
    simple(descend)).

'goal: test2' :
[?lb?, ?sf?, _0, X] -
  'goal: test2'(X, _?0, ?lb?, ?sf?):-
    simple('sort*'(_, _?0), simple(cart(natural, descend)),
    simple(cart(natural, descend))).

'goal: super' :
[?at?, _?sf?] -
  'goal: super'(_?at?, goal, _?sf?):-
    system(_?at?, system, _?sf?).

natural :
[?lb?, ?sf?, _1, _2] -
  natural(less(_, _2), ?lb?, ?sf?):-
    !,
    'natural: less'(_, _2, ?lb?, ?sf?).

[?a?, _?1?, _?s?] -
  natural(_?a?, _?1?, _?s?):-
    'natural: super'(_?a?, _?1?, _?s?).
Appendix C: Example class template programs

'natural:less' :
[?lb?, ?sf?, X, Y] -

'natural:less'(X, Y, ?lb?, ?sf?):-
  natural(X<Y, ?lb?, ?sf?).

'natural:super' :
[?at?, ?sf?] -

'natural:super'(?at?, natural, ?sf?):-
  system(?at?, system, ?sf?).

'simple' :
[?lb?, ?sf?, _1, _2] -

'simple('sort*'(_1, _2), ?lb?, ?sf?):-
  !,
  'simple:sort*'( _1, _2, ?lb?, ?sf?).

[?lb?, ?sf?, _1, _2, _3] -

'simple('insert*'(_1, _2, _3), ?lb?, ?sf?):-
  !,
  'simple:insert*'( _1, _2, _3, ?lb?, ?sf?).

'simple:sort*' :
[?lb?, ?sf?] -

'simple:sort*'( [], [], ?lb?, ?sf?):-
  !.

[?lb?, ?sf?, _1, _0, X, L] -

'simple:sort*'( [X| L], _0, ?lb?, ?sf?):-
  !,
  'simple:sort*'( L, _1, ?lb?, ?sf?),
  'simple:insert*'(X, _1, _0, ?lb?, ?sf?).

'simple:insert*' :
[?lb?, ?sf?, X] -

'simple:insert*'(X, [], [X], ?lb?, ?sf?):-
  !.

[?lb?, ?sf?, Order, X, L, D] -

'simple:insert*'(X, [D| L], [X, D| L], ?lb?, ?sf?):-
  arg(1, ?lb?, Order),
  ?:(Order, less(X, D), Order),
  !.

[?lb?, ?sf?, Order, _0, X, L, D] -

'simple:insert*'(X, [D| L], [D| _0], ?lb?, ?sf?):-
  +(arg(1, ?lb?, Order), ?:(Order, less(X, D), Order)),
Example queries:

:- goal: test1([1, 2, 3, 0, 4], X)

No1  X=[4, 3, 2, 1, 0]

:- goal: test2([(3,4), (1,6), (3,2), (10,5)], X)

No1  X=[(1,6), (3,4), (3,2), (10,5)]
Glossary

condition
A condition is an arbitrary combination of disjunctions, conjunctions and negations of predications, optionally prefixed with a label (in which case it is called a labelled condition).

conditional equality
A conditional equality is a statement of the form:

\[ f(t_1, \ldots, t_n) = G : C_1, \ldots, C_m. \quad n, m \geq 0 \]

This states that the terms \( f(t_1, \ldots, t_n) \) and \( G \) are equal whenever the conditions \( C_1 \) hold.

canonical term
See reducible expression

class rule
Class rules are used to describe inheritance between Class template programs. A Class rule is written as:

\[ l(l_1, \ldots, l_l) <= m(m_1, \ldots, m_m) \]

or

\[ l(l_1, \ldots, l_l) << m(m_1, \ldots, m_m) \]

definitional sentence
A definitional sentence of a function \( f \) is a sentence of the form

\[ f(t_1, \ldots, t_n) = R \iff f^*(t_1, \ldots, t_n, R) \]

This sentence states that a function \( f \) has a value iff the relation \( f^* \) is defined.

functional
A functional is a function valued function. Typically the right hand side of its defining equation consists of a lambda expression or set abstraction.
**label term**

A *label term* is a term which is associated with a class body, class rule or a labelled condition.

**labelled clauses**

A *labelled clause* is a sentence of the form

\[
\text{lab: head } \leftarrow \text{lab}_1\cdot c_1 \land \ldots \land \text{lab}_n\cdot c_n \quad n \geq 0
\]

where \text{lab}_i are label terms and \text{head}, c_i are conditions.

**labelled condition**

A *labelled condition* consists of a conventional condition (including negated conditions) prefixed by a label term. It is written as:

\[
\ldots, \text{lab}(a_1, \ldots, a_k):\text{pred}(t_1, \ldots, t_n), \ldots
\]

**labelled expression**

A *labelled expression* consists of an expression which is prefixed by a label term. It is written as:

\[
\text{lab}(a_1, \ldots, a_k):E
\]

**lambda expression**

A *lambda expression* is a term of the form:

\[
\lambda(t_1, \ldots, t_n)\cdot E := C
\]

where \(E\) has the same form as the right hand side of an equation and \(C\) is an arbitrary conjunction which acts as a constraint on the values of variables in much the same manner as the condition part of the conditional equation. Lambda expressions denote functions.

**narrowing**

*Narrowing* is a combined unification expression reduction algorithm in which canonical terms are unified and reducible terms are evaluated.
predication

A *predication* is a formula of the form \( p(t_1, \ldots, t_n) \) where \( p \) is a predicate symbol and the \( t_i \) are all terms. Formulae of this form are also variously called *atoms* or *literals*. However, particularly in the context of programming languages, these terms are potentially ambiguous so we borrow the notion of predication from [Robinson'79].

query

A *query* is an expression to be evaluated by a compiler system. The general form of a query is:

\[
\text{expression}?
\]

where *expression* is either a functional expression, a predication or a set abstraction:

\[
\{(t_1, t_2, \ldots, t_n) | C\}?
\]

This latter form is interpreted as a request to print the denoted relation: i.e. to find all the answers to the condition \( C \) and print the values of \( (t_1, t_2, \ldots, t_n) \).

reducible expression

A *reducible expression* is a term of the form \( f(t_1, t_2, \ldots, t_n) \) where there is an equation of the form:

\[
f(a_1, a_2, \ldots, a_n) = t :- B
\]

or where one or more of \( t_i \) are reducible.

An *irreducible* expression is a term of the form \( f(t_1, t_2, \ldots, t_n) \) where there is no equation for \( f \), furthermore \( t_i \) are all also irreducible. A *canonical* term is irreducible.

self

The *self* keyword refers to the label of a class template program before the application of any class rules.
set abstraction

A set abstraction is an expression of the form \( \{ (t_1, \ldots, t_n) \mid C \} \) where the \( t_i \) are terms and \( C \) is a condition. This expression denotes the set/relation of tuples \( (t_1, \ldots, t_n) \) such that the condition \( C \) is true. Set abstractions are the relational equivalent of \( \lambda \)-expressions.

stratified program

A stratified program is a set of clauses which can be divided into ordered subsets in such a way that for each predicate symbol \( p \) conditions involving the negation of \( p \) only occur at greater levels, never at the same level (or lower) as any level which includes the defining clauses for \( p \). The concept of stratification is important to establish the safety of using negation-as-failure.

super

The super keyword refers to the inherited only set of atomic consequences of a class template program.

term

The definition of term is built inductively as follows:

i) All constants are terms.
ii) All variables are terms
iii) If \( t_i \) are terms, then so is \( t_1(t_2, \ldots, t_n) \) a term.

theory

A theory of a set of axioms is the set of sentences which are logical consequences from the axioms.

variable call

A variable call is a condition in a clause or equation which is represented in the text by a variable. The most famous example of this is in the implementation of negation-as-failure:

\[
\text{not}(A) :- A,!,\text{fail}.
\]

\[
\text{not}(A).
\]

variable function

A variable function symbol is a function symbol of a term which is a variable in
the program text. Usually used to denote the application of a functional to arguments.

variable predicate

A predication where the predicate symbol is a variable is said to have a variable predicate.
Bibliography


[Cambridge Consultants Ltd, Logic Programming Associates Ltd], 1984, “A design study and evaluation of an APM processor”, CCL report C2033

[E. Charniak, D. McDermott], 1985, “Introduction to Artificial Intelligence”, Addison-Wesley.


[T. Duncan], 1988, "los — an implementation of Logic and Objects", AIAI, Edinburgh.


[H. Gallaire], 1986, "Merging Objects and logic programming: Relational semantics". AAAI'86


[J.A. Goguen & J. Meseguer], 1985 draft, "Object oriented programming as reflective equational programming".

[A. Goldberg, Robson D], 1983, "Smalltalk-80: The language and its implementation". Addison-Wesley.

[S. Gregory], 1987, "Parallel logic programming in Parlog", Addison-wesley.


[K. Kahn, M. Miller, D. Bobrow], 1986, "Objects in concurrent logic programming languages". Proceedings of OOPSLA'86.


[R.A. Kowalski], 1982, "The Use of metalanguage to assemble object level programs and abstract programs", Imperial College, London.


[F.G. McCabe], 1984, "The Σ Machine - a specification", Internal report, Imperial


[R. Montegue], 1973, "The proper treatment of quantification in English", Approaches to Natural Language, K.J.J. Hintikka, et.al., (eds), Dordecht, pp 221-242


[P. Roussel], 1975, "PROLOG - Manuel de reference et d'Utilisation", Groupe d'Intelligence Artificielle, UER de Luminy, Université d'Aix-Marseille II.


[D.H.D. Warren], 1977, "Implementing Prolog - compiling predicate logic programs", DAI internal reports 39 & 40, Department of Artificial Intelligence, University of Edinburgh.

