An Introduction to Progol

Sam Roberts

January 21, 1997
Contents

1 Introduction 3
  1.1 What Progol is 3
    1.1.1 Machine Learning 3
    1.1.2 Inductive Logic Programming 4
  1.2 What Progol is not 5
  1.3 How to obtain Progol 5
  1.4 Overview 6

2 An Introduction to Prolog 7
  2.1 Facts 7
  2.2 Variables 8
  2.3 Rules 9
  2.4 Recursion 11
  2.5 Lists 12

3 An example of Progol 14
  3.1 The Problem 14
    3.1.1 A Positive Example 14
    3.1.2 Head Mode Declarations and Types 15
    3.1.3 Testing for correctness with ! 16
    3.1.4 The listing facility 17
    3.1.5 Body Mode Declarations 17
    3.1.6 The modes facility 18
    3.1.7 Negative Examples 19
    3.1.8 Generalising 20
    3.1.9 Batch mode 21

4 Mode Declarations 23
  4.1 *, -, and # types 23
    4.1.1 Input and output variables 24
  4.2 Mode Declaration Recall 25
5 Parameter Settings 26

6 Testing Procedures 29
   6.1 $N > 100$: test/1 ................................. 29
   6.2 $N < 10$: leave-one-out with leave/1 ............. 32
   6.3 $10 < N < 100$: 10-fold cross validation ............. 33

7 Learning from positive data 34
   7.1 The posonly parameter and range-restricted clauses .... 34

8 Other commands 36
Chapter 1

Introduction

1.1 What Progol is

- Progol is a Machine Learning procedure
- Progol is a form of Inductive Logic Programming

1.1.1 Machine Learning

How is learning achieved in humans? There are many forms of human learning, stretching from “learning by being told” to “learning by discovery”. At the first extreme, the teacher explicitly tells the learner everything which is to be learned. This can be compared in a machine learning context to computer programming. At the other extreme, the learner autonomously discovers new facts, either by observing the environment in an unstructured fashion (as a child might do) or by planning and performing carefully constructed experiments (as a scientist might do).

Between these two extremes lies a third form of learning: learning from examples. The teacher provides examples, and the learner abstracts what is common to these examples to find a generalization. The teacher can help this process by providing a range of appropriate examples.

This is the way which Machine Learning procedures work, and Machine Learning techniques have produced many impressive results. For example, many tasks such as patient diagnosis, predicting properties of chemical compounds, and designing computer circuits can be learned. Machine Learning techniques are most appropriate where it is easier for the teacher to provide a range of good examples than a complete and explicit theory.
1.1.2 Inductive Logic Programming

All these tasks can be formulated as learning concepts from examples; the learning system develops its concepts (say, its concept of what makes a diabetic patient) by generalizing from training examples (of past diabetic and non-diabetic patients) provided by the teacher. This concept can be used to predict future examples (in this case, by diagnosing incoming patients).

Declarative Knowledge

Inductive Logic Programming has been defined as the intersection of Machine Learning and Logic Programming. This means that the examples which are given to the learning system are expressed by the teacher in a logic programming language such as Prolog, an introduction to which is contained in chapter 2. Moreover, the concepts which the learning system develops from the examples are also expressed in the same language.

This can be an advantage of Inductive Logic Programming over other forms of Machine Learning. Let us compare the situation with another Machine Learning technique, neural networks. A neural network is trained on examples of a concept and synaptic strengths between neurons are strengthened or weakened as a result of this training until the network can identify future examples of the concept with high regularity. The network has learnt a procedure for identifying new examples. One problem with this situation is that it is often a difficult and complex task to determine how exactly this procedure is working; no explicit rules are generated by the network. The knowledge which the network has gained is known as procedural knowledge.

By contrast, examples are given to an Inductive Logic Programming system in a simple logic programming language. The system generalizes these examples and produces an explicit, general rule, also expressed in a simple and clear form, which can be used to identify future examples. The system has not procedural but declarative knowledge.

This distinction is analogous to the distinction between knowing how and knowing that. Someone might, for example, know how to distinguish evergreen from deciduous trees – he may have learned the distinction by example as a child. However, he may nevertheless not know that a tree is deciduous if it loses its leaves in winter and evergreen if it is does not.

Background Knowledge

Inductive Logic Programming also makes use of background knowledge, also supplied by the teacher\(^1\) and expressed in a logic programming language. This is also an advantage; it can be viewed as mixing "learning by being told" and

\(^1\)In fact, this is not always the case. Prolog can use information it has previously learned as background knowledge for future learning; this is known as incremental learning.
“learning by examples”. It was said earlier that Machine Learning approaches are most appropriate when it is easier for the teacher to provide good examples than a complete, explicit theory. In some circumstances, of course, it may be possible to provide a partial theory to aid the process of learning by example. Inductive Logic Programming systems can make use of this background knowledge in constructing general rules.

1.2 What Progol is not

- Progol is not a programming language
  - although it is an extension of the programming language Prolog.
- Progol is not an automatic programming tool
  - although it can aid the development of logic programs.
- Progol is not a neural net package

1.3 How to obtain Progol

Progol is freely available for academic research. Progol is also available under license for commercial research. To apply for such a license, please write to steve@comlab.ox.ac.uk.

Progol is available from Oxford University Computing Laboratory’s ftp site. To obtain it, type the following.

$ ftp ftp.comlab.ox.ac.uk

When asked, enter username anonymous and your complete e-mail address as password. Then type the following at the ftp> prompt.

ftp> get pub/Packages/ILP/progol4.2/

and

ftp> quit

You should now have a directory named progol4.2/, containing four files: expand, retract, README, and progol4-2.tar.gz. Now type the following shell command.

$ expand

This should produce the subdirectories examples/ and source/, and compile progol in the subdirectory source/.

Include Progol’s source directory in your .login file to allow you to run Progol as a command.
1.4 Overview

Chapter 2 is an introduction to simple Prolog syntax. Prolog is the logic programming language in which examples, background knowledge and the general rules which Progol constructs are expressed. Chapter 3 provides a worked example of how to use Progol to solve a simple problem involving concept learning. Mode declarations are introduced in this chapter without great discussion, but because of their importance they are discussed in more detail in Chapter 4. User defined parameters are the topic of Chapter 5, and Chapter 6 concerns a variety of testing procedures. The special area of learning from positive data only is discussed in Chapter 7, and in Chapter 8 there is an explanation of all Progol’s commands and facilities.
Chapter 2

An Introduction to Prolog

Prolog is a programming language for symbolic computation and reasoning. It is particularly useful for tasks which involve the representation of facts and rules. A simple task involves reasoning about a family tree, shown below in Figure 2.1.

2.1 Facts

The fact that Charles is a parent of William is represented in Prolog as follows.

\[
\text{parent(charles,william).}
\]

Thus the entire family tree can be represented in Prolog as follows.

\[
\begin{align*}
\text{parent(george,elizabeth).} \\
\text{parent(edward,diana).} \\
\text{parent(frances,diana).} \\
\text{parent(elizabeth,charles).} \\
\text{parent(anne,charles).} \\
\text{parent(anne,charles).} \\
\text{parent(diana,william).} \\
\text{parent(charles,william).}
\end{align*}
\]

This program contains nine clauses, concerning the parent relation. It is a relation between two people; we say it has \textit{arity 2}, and write parent/2. Another word commonly used in place of “relation” is “predicaté”. Once Prolog has been given this program, it can be asked questions concerning the parent relation. The following is such a question.

\[
\text{?- parent(charles,william).}
\]
Figure 2.1: A family tree

Prolog finds this fact in the program and replies as follows.

yes

We can also ask Prolog the following question.

?- parent(philip,william)?

to which Prolog answers as follows.

no

Prolog also answers no to the following question.

?- parent(charles,hirohito)?

This is because it has never heard of Hirohito.

2.2 Variables

All the names in the family tree program are written with a small letter. This is because they represent constants. All constants in Prolog must start with a small letter. Variables are represented in Prolog starting with a capital letter.

For example, we can ask Prolog the following question.

?- parent(diana,X)?
This meaning “Who is Diana a parent of?”, to which Prolog replies as follows.

\[ X = \text{william} \]

Some questions may have more than one answer, for example “Who is Philip a parent of?”. All the answers which Prolog can find can be obtained by typing semi-colons after answers.

\[ \text{?- parent(philip,}X)\]?

\[ X = \text{charles;} \]

\[ X = \text{anne;} \]

no

Prolog answers no when it cannot find any more answers.

### 2.3 Rules

It is not only simple facts such as that Charles is a parent of William which are expressible in Prolog. Also expressible are rules such as that if \( X \) is a parent of \( Y \) then \( Y \) is a child of \( X \). This is represented in a Prolog program by the following clause.

\[ \text{child}(Y,X):= \text{parent}(X,Y). \]

“\( \text{child}(Y,X) \)” is the head of the clause and “\( \text{parent}(X,Y) \)” is the body of the clause.

Let us add the above clause to the family tree program. We can now ask Prolog questions concerning our new child relation. For example, to the question

\[ \text{?- child(william,diana)?} \]

Prolog replies:

yes

and to the question

\[ \text{?- child}(X,elizabeth)? \]

Prolog replies
X = charles;
X = anne;

no

The use of the child rule is much simpler than adding another complete set of child facts to the program.

Prolog can handle more complicated rules than this. The grandparent relation can be represented in Prolog by the clause:

\[
\text{grandparent}(X,Z) :\text{= parent}(X,Y), \text{parent}(Y,Z).
\]

The comma between the two conditions in the body indicate that both conditions must be true.

Prolog, given the question

\[\text{\texttt{\textbf{?- grandparent(X,william)}}}\]

will now answer

X = edward;
X = frances;
X = elizabeth;
X = philip;

no

Let us add some more facts to our family tree program, to represent the sex of the family members.

male(george).
male(edward).
female(frances).
female(elizabeth).
male(philip).
female(diana).
male(charles).
female(anne).
male(william).

We can then add the following rule for father.

\[
\text{father}(X,Y) :\text{= parent}(X,Y), \text{male}(X).
\]
2.4 Recursion

So far the family tree program contains facts concerning the parent relation and the sex of family members, as well as rules for child, grandparent, and father. Many other rules could be similarly represented, such as mother, grandmother, great-grandparent etc.

Not all such relations can be so simply represented, however. Consider what is involved in the ancestor relation. \( X \) is a ancestor of \( Y \) if \( X \) is a parent of \( Y \), or \( X \) is a grandparent of \( Y \), or . . . . A first attempt at a program representing the ancestor relation might then be as follows.

\[
\text{ancestor}(X,Z) :\text{ parent}(X,Z).
\]

\[
\text{ancestor}(X,Z) :\text{ parent}(X,Y),\text{parent}(Y,Z).
\]

\[
\text{ancestor}(X,Z) :\text{ parent}(X,Y1),\text{parent}(Y1,Y2),\text{parent}(Y2,Z)
\]

It is clear that this first attempt program will only get ancestors up to great-great-grandparents. Moreover, it is clear that if we attempted to extend this program in the obvious way to catch the general idea of an ancestor, the program would have to be infinite.

Fortunately there is another way in Prolog of constructing the ancestor relation. The idea is to define it in terms of itself. We have the following two clauses.

\[
\text{ancestor}(X,Y) :\text{ parent}(X,Y).
\]

\[
\text{ancestor}(X,Z) :\text{ parent}(X,Y),\text{ancestor}(Y,Z).
\]

Such definitions are called recursive clauses. With this addition to the family tree program, Prolog will now answer as follows.

\[
\text{?- ancestor}(X,\text{charles})?
\]

\[
X = \text{elizabeth};
\]

\[
X = \text{philip};
\]

\[
X = \text{george};
\]

no

\[
\text{?- ancestor}(\text{george},X)?
\]

\[
X = \text{elizabeth};
\]
\[ X = \text{charles}; \]
\[ X = \text{anne}; \]
\[ X = \text{william}; \]
\[ \text{no} \]

Recursive clauses are vital to Prolog programming and occur in almost all programs of complexity.

### 2.5 Lists

Lists are a simple data structure used commonly in Prolog programming. A list is a sequence of items, for example \texttt{elizabeth, charles, william, anne}. This list is represented in Prolog as:

\[
[\text{elizabeth, charles, william, anne}]\]

A special case, the \textit{empty list}, or list containing no items, is represented as follows.

\[
[]
\]

It is often useful in Prolog to split a list into two parts:

- the first item, or \textit{head} of the list, and
- the rest, or \textit{tail} of the list.

There is a special notation in Prolog for this purpose. Lists can also be written in the following form.

\[
[\text{Head} | \text{Tail}]\]

It should be remembered that \texttt{Tail} is another list. Thus the following lists are all equivalent.

\[
[\text{elizabeth, charles, william, anne}],
[\text{elizabeth} | [\text{charles, william, anne}]],
[\text{elizabeth, charles} | [\text{william, anne}]],
[\text{elizabeth, charles, william} | [\text{anne}]],
[\text{elizabeth, charles, william, anne} | []] \]

As an example of the use of list notation (and a further example of recursion), consider the \textit{member} relation. Observe that \( X \) is a member of a list \( L \) if
\begin{itemize}
\item $X$ is the head of $L$, or
\item $X$ is a member of the tail of $L$.
\end{itemize}

Clauses for the membership relation can therefore be written as follows.

\begin{verbatim}
member(X, [X | Tail]).
member(X, [Head | Tail]) :- member(X, Tail).
\end{verbatim}

Lists are especially useful when combined with recursion. Let us add the following facts to the family tree program.

\begin{verbatim}
enjoys(william, tennis).
enjoys(charles, polo).
enjoys(philip, grouse-shooting).
enjoys(anne, equestrianism).
\end{verbatim}

We can then define the recursive clause

\begin{verbatim}
enjoy([], []).
enjoy([Person | Others], [Sport | Sports]) :-
enjoys(Person, Sport),
enjoy(Others, Sports).
\end{verbatim}

Prolog will now give the following answer.

\begin{verbatim}
\texttt{|- enjoy([anne, charles, william, philip], X)}?
\end{verbatim}

\begin{verbatim}
X = [equestrianism, polo, tennis, grouse-shooting]
\end{verbatim}
Chapter 3

An example of Progol

This chapter contains a worked example of how to use Progol to solve a simple problem. Many ideas are introduced in this chapter without much comment; they are covered in more detail in later chapters. This chapter is meant as a broad introduction to the sort of problem with which Progol can help, and to the sort of solution which it provides.

3.1 The Problem

We are given a sequence of trains (see fig 3.1). Each train has attached a number of cars, each of which may have a number of different properties such as being long or short, having a roof or no roof, and having a shape painted on the side. In addition, we are given that each train is either travelling east or travelling west. The problem is to find a rule which will predict, from the properties of its cars, in which direction a train is travelling. Before seeing how Progol deals with this problem, it is worthwhile attempting to solve this problem manually to get a measure of its difficulty.

3.1.1 A Positive Example

The first thing is to start Progol. This is done (assuming the Progol source directory has been added to the login file) by typing the following.

$ progol

Progol is now running in interactive mode, and a “|-” prompt is seen.

The first positive example can now be given to Progol.

|- eastbound(train1).

Progol responds with the following.
This is saying that Progol has added the clause to its stock of examples, and has tested the examples to see if they are contradictory. They are not, since we have so far only told Progol the one fact that train number one is eastbound.

### 3.1.2 Head Mode Declarations and Types

We now need a *mode declaration*. Mode declarations are used by Progol to guide the process of constructing a generalization from its examples. We type the following.

```prolog
?- modelh(1, eastbound(+train)).
```

Mode declarations are discussed in more detail in Chapter 4. As we have said in Chapter 1, the general rules which Progol constructs are also expressed in Prolog clauses, which have *a head* and *body*. This mode declaration says that the general rules may have heads (it is a model declaration; we will come across a corresponding modeb declaration soon for clause bodies) containing
eastbound(X), where X is a variable of type train. The number 1 is called the recall and is discussed in Chapter 4.

Prolog responds to the mode declaration as follows.

?- modeh(1,eastbound(+train))?  
yes
[?- modeh(1,eastbound(+train))? - Time taken 0.02s]
?-  

Having said that eastbound is applied to things of type train, we must ensure that train number one is indeed of type train. We do this by typing the following.

?- train(train1).

3.1.3 Testing for correctness with !

Prolog now has a minimal amount of information about the sequence of trains and how to construct general rules concerning whether or not they are eastbound. To check whether or not we have entered this information correctly, we can type the following.

?- eastbound(train1)!

This instructs Prolog to construct the most specific clause from the example given and the mode declarations. The most specific clause is an object which Prolog uses in the process of constructing general rules from examples. We will not go into details of the theory underlying its use, but we can use it as a convenient test to ensure that we are entering the information in roughly the correct manner. In any case where we have entered a single positive example of a property p, along with a head mode declaration and type, we expect the most specific clause to be p(A). Thus, if all is correct, Prolog will respond as follows.

?- eastbound(train1)!
[Testing for contradictions]
[No contradictions found]
[Most specific clause is]
eastbound(A).

[eastbound(train1)! - Time taken 0.05s]
?-  

16
3.1.4 The listing facility

This reassures us that we are entering the information in a correct manner, and we can continue to add more. We can add some more positive examples first.

\[- \text{eastbound(train2)}.
\]
\[- \text{eastbound(train3)}.
\]
\[- \text{train(train2)}.
\]
\[- \text{train(train3)}.
\]

Let us check what information Progol now knows about the sequence of trains. We can do this by using the listing facility. We type the following.

\[- \text{listing(eastbound/1)}?\]
\text{eastbound(train1)}.
\text{eastbound(train2)}.
\text{eastbound(train3)}.
\[\text{[Total number of clauses = 3]}\]
yes
\[\text{[}\text{listing(eastbound/1)}? - \text{Time taken 0.00s}]\]
\[-
\]

The listing facility presents a complete rundown of all clauses Progol currently knows with heads containing eastbound. The /1 in eastbound/1 is its arity (see Chapter 2), which must accompany a predicate when using listing. We can do the same for train/1. The listing facility is used regularly to check what Progol currently knows.

3.1.5 Body Mode Declarations

We can add more information about the trains and their cars. Let us adopt the convention that the \( m \) th car after train number \( m \) is called car\(-m\).

\[- \text{nextcar(train1,car1_1)}.
\]
\[- \text{nextcar(train2,car2_1)}.
\]
\[- \text{nextcar(train3,car3_1)}.
\]

We now need a body mode declaration. We type the following.

\[- \text{modeb(1,nextcar(+train,-car))}?
\]
This mode declaration says that the general rules which Progol constructs may have bodies containing predicates of the form \texttt{nextcar}(X,Y) where \( X \) is a variable of type \texttt{train} and \( Y \) is a variable of type \texttt{car}. Again, the 1 is the \texttt{recall} and is explained in Chapter 4, as is the use of + and - with types.

We also remember to add the correct type information.

\[
\texttt{|- car(car1_1).} \\
\texttt{|- car(car2_1).} \\
\texttt{|- car(car3_1).}
\]

### 3.1.6 The modes facility

We can also add information about the shapes on the cars. We type the following. (The use of \# with types is explained in Chapter 4.)

\[
\texttt{|- :- modeb(1,shape(+car,#shape))?} \\
\texttt{|- shape(car1_1,circle).} \\
\texttt{|- shape(car2_1,circle).} \\
\texttt{|- shape(car3_1,circle).} \\
\texttt{|- shape(circle).}
\]

We have now entered all the information relating to the eastbound trains. We have already introduced the \texttt{listing} facility for checking what Progol knows about a particular predicate; there is a similar method of checking what information Progol has about the current mode declarations. We simply type the following.

\[
\texttt{|- modes?}
\]

Progol replies to this as follows.

\[
\texttt{|- modes?}
\texttt{Head modes}
\texttt{ mode(1,eastbound(+train))}
\texttt{Body modes}
\texttt{ mode(1,nextcar(+train,-car))}
\texttt{ mode(1,shape(+car,#shape))}
\texttt{yes}
\texttt{[;:- modes? - Time taken 0.00s]}
\]
3.1.7 Negative Examples

The only task remaining is to enter information relating to the trains which are not eastbound, i.e. the negative examples of `eastbound`. Negative examples are given to Prolog as follows.

```
|  | - eastbound(train4).
|  | - eastbound(train5).
|  | - eastbound(train6).

The fact that the example is negative rather than positive is marked by the occurrence of `:-` before the example.

We must also enter additional type and background information for the new trains and cars.

```
|  | - train(train4).
|  | - train(train5).
|  | - train(train6).
|  | - nextcar(train4,car4_1).
|  | - nextcar(train5,car5_1).
|  | - nextcar(train6,car6_1).
|  | - car(car4_1).
|  | - car(car5_1).
|  | - car(car6_1).
|  | - shape(car4_1,square).
|  | - shape(car5_1,square).
|  | - shape(car6_1,square).
|  | - shape(square).
```
3.1.8 Generalising

Having entered all the information we have relating to the trains, we can ask
Propal to generalise from these examples and form a more general rule. We type
the following.

!- generalise(eastbound/1)?

Propal replies to this as follows.

!- generalise(eastbound/1)?
[Generalising eastbound(train1).]
[Most specific clause is]

eastbound(A) :- nextcar(A,B), shape(B,circle).

[C:-0,3,3,0 eastbound(A).]
[C:-1,3,3,0 eastbound(A) :- nextcar(A,B).]
[C:1,3,0,0 eastbound(A) :- nextcar(A,B), shape(B,circle).]
[3 explored search nodes]
f=1,p=3,n=0,h=0
[Result of search is]

eastbound(A) :- nextcar(A,B), shape(B,circle).

[3 redundant clauses retracted]
eastbound(A) :- nextcar(A,B), shape(B,circle).
[Total number of clauses = 1]

yes
[!- generalise(eastbound/1)? - Time taken 0.10s]
!-

This requires some explanation. What Propal does when asked to construct
a general rule can be divided into three parts. Without going into detail, the
first part, which takes up to the third line of the above output, is to construct the
most specific clause of the first positive example. In this case, the first positive
example is eastbound(train1), and its most specific clause is eastbound(A) :-
nextcar(A,B), shape(B,circle).

The second part is to make new clauses by combining the predicates con-
tained in the most specific clause. It compares these new clauses by considering
how many of the original examples they explain (positive and negative). When
it finds a clause which explains most of the positive examples (and none of the
negative ones), Propal selects this clause as its general rule. In this case, the
result is in fact the most specific clause itself.
The third part is to add this general rule to its information concerning the trains, and remove any of the original positive examples which are now redundant. In this case, the new clause explains all of the original positive examples, and all three are removed. Progol reports the current state of its knowledge concerning the \texttt{eastbound} predicate, which is now that a train is heading east if its first car has a circle on it.

### 3.1.9 Batch mode

So far we have been running Progol in \textit{interactive} mode. This is especially useful for small amounts of data and for testing a small part of larger data sets to see if we are entering the data in a sensible way.

An alternative is to give Progol all the data at once, rather than in the piecemeal way we have been doing so far. We can enter the data into a text file, using an editor such as \texttt{emacs}. A file containing the data for the trains example is given in figure 3.2.

We can then run Progol on this dataset from the command line as follows.

\begin{verbatim}
$ progol trains
\end{verbatim}

This causes Progol to generalise every predicate with a \texttt{modeh} declaration in the file. In this case, the only such predicate is \texttt{eastbound}, and Progol outputs the same result as when asked to \texttt{generalise(eastbound/1)} in interactive mode.
%% Mode Declarations

:- modeh(1,eastbound(+train)).
:- modeh(1,nextcar(+train,-car)).
:- modeh(1,shape(+car,#shape)).

%% Types

train(train1). train(train2). train(train3).
train(train4). train(train5). train(train6).
car(car1_1). car(car2_1). car(car3_1).
car(car4_1). car(car5_1). car(car6_1).
shape(circle). shape(square).

%% Background Information

defnextcar(train1,car1_1). nextcar(train2,car2_1).
defnextcar(train3,car3_1). nextcar(train4,car4_1).
defnextcar(train5,car5_1). nextcar(train6,car6_1).
defshape(car1_1,circle). shape(car2_1,circle).
defshape(car3_1,circle). shape(car4_1,square).
defshape(car5_1,square). shape(car6_1,square).

%% Positive Examples

eastbound(train1). eastbound(train2).
eastbound(train3).

%% Negative Examples

:- eastbound(train4). :- eastbound(train5).
:- eastbound(train6).

Figure 3.2: The file trains.pl
Chapter 4

Mode Declarations

Mode declarations are at the heart of Progol’s method of generalising examples, and it is important that they are understood if Progol is to learn in the most efficient way. So far we have introduced them in the context of a simple example, and have said that they are used by Progol in constructing the general rules from particular examples.

In particular, we have said that they restrict the predicates which can occur in the head and body of the general rules. Thus in our trains example, the mode declarations were as follows.

:- modeh(1,eastbound(+train))?  
:- modeb(1,nextcar(+train,-car))?  
:- modeb(1,shape(+car,#shape))?  

The first of these says that the general rules may have heads containing the predicate eastbound(X), where X is of type train. The second says that the general rules may have bodies containing the predicate nextcar(X,Y), where X is of type train and Y is of type car. The third is similar.

There are two other ways in which the mode declarations are used by Progol in constraining its search for a general rule, however; the recall and the use of +, −, and # with types.

4.1 +, −, and # types

We have seen that the first mode declaration above gives the information that the rules may have heads containing the predicate eastbound(X), where X is of type train, but the declaration actually tells us slightly more than this; because the type is +train, it says that X must be a variable of type train. Thus, we can have rules with heads containing, for example, eastbound(A), but not heads containing eastbound(train1).
Conversely, the # in the #shape type says that we must have constants, not variables, of type shape. Thus we can have rules with bodies containing shape(B,circle) but not shape(B,C).

There is, however, a complication. The -, as well as +, also requires its types to be instantiated with variables rather than constants. What is the difference?

4.1.1 Input and output variables

The simplest way to demonstrate the difference is to give examples of input and output variables in Prolog. Let us recall some of the examples of Chapter 2, in particular the member predicate, which was defined as follows.

member(X,[X|Tail]).

member(X,[Head|Tail]):- member(X,Tail).

We can ask Prolog questions such as the following.

\[ \text{member}(1,[1,2])? \]

yes

\[ \text{member}(X,[1,2])? \]

\[ X = 1; \]

\[ X = 2; \]

no

Consider what would happen if we asked Prolog the following question.

\[ \text{member}(1,X)? \]

This question is asking Prolog what lists 1 is a member of. Clearly there are an infinite number of answers, and Prolog cannot find all of them. In fact Prolog does give a series of answers, but none are of much help. The member predicate was not really written with those sorts of questions in mind; the first argument was meant to be the variable, not the second. This is expressed by saying that the first argument is an output variable and the second is an input variable. The distinction is not always clear (for example, both parent(X,william)? and parent(charles,X) are sensible questions), but many predicates may give unexpected results if questions are asked concerning them in which the variables occur in input arguments.

This sort of difficulty is prevented in Prolog by the use of + and - in conjunction with types. + types are used where there is an input argument of a predicate, and - types are used for an output argument. In cases where arguments can be both input and output, two mode declarations can be given.
4.2 Mode Declaration Recall

The number 1 is the recall of the mode declarations. The recall can be any positive whole number \( n \geq 1 \), or \( "^*" \). The recall is used as a bound on the number of alternative instantiations of the predicate. An instantiation of the predicate is a replacement of the types by either variables or constants in accordance with the \( + \), \( - \), and \( # \) information.

If we know that there are only a certain number of solutions for a particular instantiation, we can tell this to Prolog in the recall to save Prolog searching fruitlessly for further solutions. For example, we know that each train only has one car following it, so we can give the recall of the next car mode declaration as 1. If we were giving a mode declaration for the parent predicate in some problem, we might give the recall as 2, since everyone has at most two parents, and similarly 4 for a grandparent declaration.

The \( "^*" \) recall is used when there is no limit to the number of solutions for an instantiation. For example, there is no limit on the number of ancestors people have, so we might have \( "^*" \) for the recall in an ancestor mode declaration. In fact, Prolog substitutes an arbitrary large number (by default 100) for \( "^*" \) recalls, and will stop after this number of instantiations; in practice this does not cause any difficulties.
Chapter 5

Parameter Settings

There are several user settable parameters which control the way in which Progol carries out its tasks. Each is described below. Some parameters take integer values; these are marked below with “(N)”, and are set with the system predicate \texttt{set(\texttt{Parameter},Value)}?. The remainder take as values either \texttt{ON} or \texttt{OFF}. They are turned on and off with the system predicates \texttt{set(\texttt{Parameter})}? and \texttt{unset(\texttt{Parameter})}?. The values of all user definable parameters can be examined by typing \texttt{settings}?.

\textbullet \texttt{c} (N) \texttt{c} is the maximum length of (i.e., number of atoms in) the body of the general rules which Progol constructs. Thus, when Progol is considering the various clauses it can make out of combinations of predicates in the most specific clause (see p.20), it will discard any clauses of length greater than \texttt{c}. This can be useful as in many applications the most specific clause may be very large, and consideration of every combination may be a long and fruitless job. The default for \texttt{c} is 4.

\textbullet \texttt{condition} (\texttt{ON}/\texttt{OFF}) The \texttt{condition} setting is used when learning from positive data. See Chapter 7 for more details on this subject. When \texttt{condition} is set to \texttt{ON}, Progol will construct the probability distribution from the examples given. When set to \texttt{OFF}, it assumes they are drawn from a uniform distribution. The default for \texttt{condition} is \texttt{ON}.

\textbullet \texttt{cover} (\texttt{ON}/\texttt{OFF}) When set to \texttt{ON}, Progol uses \texttt{cover} searching. This is essential for learning recursive clauses. When set to \texttt{OFF}, Progol uses \texttt{implicational} searching. This is faster but recursive clauses cannot be learned. The default is \texttt{ON}.

\textbullet \texttt{h} (N) Progol constructs the most specific clause from an example using an inference rule known as \textit{resolution}. \texttt{h} is the maximum number of applications of this rule which Progol may use in deriving the most specific
clause. Often Progol will not need to use this many, but if it hits this barrier it will give a warning indicating so. The default for \( h \) is 30.

\[ \text{i (N)} \]

Progol is constrained in constructing the most specific clause not only by \( h \), but also by \( i \). \( i \) is the maximum depth of the variables which may occur in the most specific clause, where the depth \( d(v) \) of a variable \( v \) in a clause \( C \) is defined as follows.

\[
d(v) = \begin{cases} 
0, & \text{if } v \text{ is in the head of } C \\
\min_{u \in U_v} d(u) + 1, & \text{otherwise}
\end{cases}
\]

where \( U_v \) are the variables in atoms in the body of \( C \) containing \( v \).

As an example of the depth of a variable, consider the following clause.

\[ \text{eastbound(A):- nextcar(A,B), shape(B,C).} \]

In this clause, variable \( A \) has depth 0, \( B \) has depth 1, and \( C \) has depth 2.

The default for \( i \) is 3.

\[ \text{inflate (N)} \]

When Progol is searching through the combinations of predicates from the most specific clause, in order to construct its general rule, it is looking for the combination which, in a general rule, would compress the data the most — in other words, reduce the number of predicates the most. We can, in this search, give a weighting to the data or to the predicates in a general rule. This weighting is \text{inflate}, and is expressed as a percentage. Thus, for example, if Progol was constructing a general rule from 3 examples with \text{inflate} set to 100%, it would not bother learning a rule of length 3 as it would not compress the data into anything smaller. However, if \text{inflate} were set to 101%, it would — since the data now has “size” 3.03. The default for \text{inflate} is 100%.

\[ \text{memoing (ON/OFF)} \]

When Progol is searching for a general rule, it constructs many combinations and calculates various statistics for each combination. It may happen occasionally that Progol has to consider the same rule more than once. Its search will clearly be faster if it can remember the statistics from last time. This can be achieved by setting \text{memoing} to ON. The downside of setting memoing to ON is that more memory is taken up in remembering the statistics — a time/space tradeoff. The default is ON.

\[ \text{nodes (N)} \]

Progol will give up searching for a general rule after it has searched through \( N \) combinations without success. The default is 10000.

\[ \text{noise (N)} \]

We may not always require our rules to be perfect; in some cases, we may want to allow them to predict a small percentage of the negative examples. This percentage can be set by \text{noise}. The default is 0.

27
posonly (ON/OFF) When set to ON, this sets Progol to learn from positive examples only. See Chapter 7 for more details on learning from positive data. The default is OFF.

reductive (ON/OFF) We can constrain the form of the general rules in another way by setting reductive to ON. When this is done, the terms in the body of a learnt rule must be less complex than those in the head, where the complexity of a term is the number of its subterms. The default is OFF.

searching (ON/OFF) We can change the behaviour of ! by setting searching to ON. Normally ! constructs the most specific clause. When searching is ON, however, the effect of ! is the same as generalise. The default is OFF.

splitting (ON/OFF)

tracing (ON/OFF) This flag turns on the Progol tracing facility. It can also be set by the system predicate trace? and unset by notrace?. The default is off.

verbose (N) This sets the amount of output Progol gives, and takes values 0, 1 or 2. 0 gives virtually no output, just the answer itself, whereas 2 gives information, runtime statistics etc. as well. The default is 2.