CProgol4.4: a tutorial introduction

Stephen Muggleton
John Firth

Department of Computer Science,
University of York, UNITED KINGDOM.
Email: stephen@cs.york.ac.uk.

Abstract. This chapter describes the theory and use of CProgol4.4, a state-of-the-art Inductive Logic Programming (ILP) system. After explaining how to download the source code, the reader is guided through the development of Progol input files containing type definitions, mode declarations, background knowledge, examples and integrity constraints. The theory behind the system is then described using a simple example as illustration. The main algorithms in Progol are given and methods of pruning the search space of possible hypotheses are discussed. Next the application of built-in procedures for estimating predictive accuracy and statistical significance of Progol hypotheses is demonstrated. Lastly, the reader is shown how to use the more advanced features of CProgol4.4, including positive-only learning and the use of metalogical predicates for pruning the search space.

1 Introduction

The theory and implementation of the Inductive Logic Programming (ILP) system CProgol4.1 was first described in [8]. Since then a number of advances have been made over the original CProgol4.1 in systems such as CProgol4.2 [9], PProgol2.1 and PProgol2.2. The development of these systems has been informed by feedback from experiments on a variety of real-world applications [3, 6, 5, 14, 2].

This chapter describes the theory and use of CProgol4.4, a publicly distributed version of the Progol family of ILP systems. In order to follow the examples in this chapter, it is assumed that the reader is familiar with the aims of ILP, Horn clause logic and Prolog notation for clauses. It is also assumed that the reader is familiar with the concepts of deductive logic in order to understand the theoretical foundations of Progol. Furthermore it is assumed that the reader has access to a machine running UNIX with an internet connection. This will be necessary in order to obtain the software and run the examples described in the chapter.

The chapter has the following structure. Section 2 describes how to use anonymous ftp to obtain CProgol4.4, together with the associated distribution datasets and documentation. Having obtained the system, Section 3 guides the reader through the general approach that should be taken to developing a Progol input file containing examples and background knowledge. The theory behind Progol is then described in Section 4 and illustrated by a simple example. Here the main algorithms inside the system are given in quite a good deal of detail, hopefully without being too technical. The reader is then shown in Section 5 how to make use of built-in procedures which support the estimation of predictive accuracy and statistical significance of hypothesised theories. Section 6 describes how to use integrity constraints and prune statements to control search in CProgol4.4. The setting of resource bounds is discussed in Section 7. Section 8 describes the facilities available to support the debugging of a CProgol4.4 input file. Finally, Section 9 summarises the philosophy behind Progol and explains why it was designed in the way it was. Mention is made of some of the more important applications of the system to real-world problems.

2 How to obtain CProgol4.4

It should be noted that CProgol4.4 is available free for academic research and teaching. For commercial research, a license should be obtained by writing to the first author.
CProgol4.4 can be obtained in either of the following ways.

WWW: The ftp site can be accessed via the following web page:

`http://www.cs.york.ac.uk/mlg/progol.html.`

ftp: It can also be obtained directly by anonymous ftp from `ftp.cs.york.ac.uk` in the directory `pub/ML_GROUP/progol4.4` .

The ftp site contains a README file with instructions on installing the software. For this you will need to be running a UNIX (or Linux) operating system with a gcc compiler. In fact, Progol4.4 is automatically compiled in the source directory when you obtain all the files from the ftp site and run the *expand* script. This command additionally compiles a program called qsample, which is useful for randomly sampling training and test sets (instructions for running qsample are in the comments at the top of qsample.c).

To test that Progol is correctly installed after running expand, move to the source directory and issue the following command from the UNIX prompt.

```
$ progol
```

In response you should see the following.

```
Progol Version 4.4

|-
```

To see the list of available commands type the following.

```
|- help?
```

Note that in interactive mode Progol is similar to a Prolog interpreter. In fact, Progol uses the Edinburgh Prolog syntax and contains almost all of the built-in predicates used in Clocksin and Mellish's introduction to Prolog [1]. One immediate difference however is the use of '?' at the end of queries instead of '.'. Any statement ending with a '.' is asserted into the clause-base.

You can get one-line description of Progol's built-in commands using the command help in the following fashion.

```
|- help(tell/1)?
```

To quit Progol type either an end-of-file (usually Control-D) or 'quit?'.

For the following, you should include the source directory as part of the path description in your .login file in order to be able to use the 'progol' command in other directories.

3 Developing an input file for CProgol4.4

As with all ILP systems, Progol constructs logic programs from examples and background knowledge. For instance, given various examples of "aunt_of"

```
aunt_of(jane,henry).
aunt_of(sally,jim).
..
```

and background knowledge concerning "parent_of", "father_of", "mother_of" and "sister_of"

```
parent_of(Parent,Child) :- father_of(Parent,Child).
parent_of(Parent,Child) :- mother_of(Parent,Child).
..
father_of(sam,henry).
..
mother_of(sarah,jim).
..
sister_of(jane,sam).
sister_of(sally,sarah).
..
```

together with a list of types and mode declarations, Progol can construct a definition for the aunt_of predicate. In order to do this Progol needs to be told the following information about the learning task.

3.1 Types

These describe the categories of objects (numbers, lists, names, etc.) in the world under consideration. In this example we only need the type *person*, since all objects in the given relations are of this type.

```
person(jane).
person(henry).
person(sally).
person(jim).
..
```

This simply states that all of the elements involved satisfy the unary predicate *person*. In other words they have type *person*.

3.2 Modes

These describe the relations (predicates) between objects of given types which can be used either in the head (*modeh* declarations) or body (*modeb* declarations) of hypothesized clauses. Modes also describe the form these atoms can take within a clause. For the head of any general rule defining *aunt_of* we might give the following head mode declarations

```
:- modeh(1,aunt_of(+person,+person))?
```

The declaration states that head atom has predicate symbol *aunt_of* and has 2 variables of type *person* as arguments. The '+' sign indicates that an argument is an input variable. A '-' sign would indicate an output variable, and a '#' would indicate that a constant should be placed at the position in the hypothesis. Thus the declaration above indicates that

```
aunt_of(X, Y)
```

is allowed as the head atom of a general definite clause, and that goals calling this clause should bind both variables.

All modeh (and modeb) declarations contain a number called the recall. This is used to bound the number of alternative solutions for instantiating the atom. Here it is 1 because the aunt_of predicate gives a unique answer (yes or on) when given two input arguments. For a predicate such as *square_root* the recall would be 2 since a number has at most two square roots. If in doubt use a large number or '*' as the recall. The latter means no limit to the number of instantiations that might be made but, in practice, Progol will set its own upper limit of 100.

For atoms in the body of a general rule, body mode declarations must be given as follows.

```
:- modeb(*,parent\_of(-person,+person))?
:- modeb(*,parent\_of(+person,-person))?
:- modeb(*,sister\_of(+person,-person))?
```

The first of these declarations can be used to add *parent_of* atoms to the body of a hypothesis which introduce one or more parents of a given child. Similarly, the second allows *parent_of* to be used in the body to find one or more children of a given parent and the last can be used in the body to find one or more sisters of a given individual.

3.3 Settings

These describe some of the runtime parameter settings that Progol uses. For instance, in the following

```
:- set(posonly)?
```

posonly turns on the positive-only evaluation mechanism [9]. This allows the user to avoid incorporating negative examples, which are often unnatural to define, and also often unavailable in real-world domains.

All the other settings in Progol are described in the manual that is part of the source distribution.

3.4 The complete example

The full listing of the example follows. This is similar to the file `aunt.pl` found under the directory `examples4.4` in the Progol4.4 distribution (see Section 2).

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Learning aunt_of from parent_of and sister_of.

% Settings

:- set(posonly)?

% Mode declarations

:- modeh(1,aunt_of(+person,+person))?
:- modeb(*,parent_of(-person,+person))?
:- modeb(*,parent_of(+person,-person))?
:- modeb(*,sister_of(+person,-person))?

% Types

person(jane).
person(henry).
person(sally).
person(jim).
person(sam).
person(sarah).
person(judy).

% Background knowledge

parent_of(Parent,Child) :- father_of(Parent,Child).
parent_of(Parent,Child) :- mother_of(Parent,Child).
```

```
father_of(sam,henry).

mother_of(sarah,jim).

sister_of(jane,sam).
sister_of(sally,sarah).
sister_of(judy,sarah).

% Examples

aunt_of(jane,henry).
aunt_of(sally,jim).
aunt_of(judy,jim).
```

Note that a line beginning with the '%' character indicates a comment. Negative examples of *aunt_of* could have been included as follows.

```
:- aunt_of(henry,sally).
:- aunt_of(judy,sarah).
```

Note the use of ':-' to indicate negation.

3.5 The output

Assuming that the above listing is in a file called *aunt.pl* say, it would be given to Progol by typing the following on the command line.

```
progol aunt
```

Alternatively the session might be interactive in which case Progol would be executed with no arguments. The user could then type in modes, types and clauses or use Prolog input routines such as *consult* to read in files. See the manual page for further details. The output of executing the above command follows.

```
CProgol Version 4.4

[Noise has been set to 100%]
[Example inflation has been set to 400%]
[The posonly flag has been turned ON]
[:- set(posonly)? - Time taken 0.00s]
[:- modeh(1,aunt_of(+person,+person))? - Time taken 0.00s]
[:- modeb(100,parent_of(-person,+person))? - Time taken 0.00s]
[:- modeb(100,parent_of(+person,-person))? - Time taken 0.00s]
[:- modeb(100,sister_of(+person,-person))? - Time taken 0.00s]
[Testing for contradictions]
[No contradictions found]
```

```
[Generalising aunt_of(jane,henry).]
[Most specific clause is]

aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).

[Learning aunt_of/2 from positive examples]
[C:-0,12,11,0 aunt_of(A,B).]
[C:6,12,4,0 aunt_of(A,B) :- parent_of(C,B).]
[C:6,12,3,0 aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).]
[C:6,12,3,0 aunt_of(A,B) :- parent_of(C,B), sister_of(A,D).]
[C:4,12,6,0 aunt_of(A,B) :- sister_of(A,C).]
[5 explored search nodes]
f=6,p=12,n=3,h=0
[Result of search is]

aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).

[3 redundant clauses retracted]

aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).

[Total number of clauses = 1]

[Time taken 0.02s]
```

When Progol is given a file as input, it tries to develop a general rule for all predicates appearing in head mode declarations. In this case the only such predicate is *aunt_of*. It first ensures that the asserted clauses are consistent. In this case it reports that no contradictions were found so the input is consistent. It then constructs the most specific clause of the first positive example in the input. The first positive example is

```
aunt_of(jane,henry).
```

and its most specific clause is

```
aunt_of(A,B) :- parent_of(C,B), sister_of(A,C).
```

It then generates new clauses from this most specific clause and sees how many of the examples they prove. Here the clause

```
aunt_of(A,B).
```

explains 3 positive examples (400% inflation during positive-only learning and 3*4=12 is the second number in the list) and 11 random instances (the third element in the list). Similarly

```
aunt_of(A,B) :- parent_of(C,B).
```

explains the same number of positive examples but fewer (5) randomly constructed instances, and thus has a better measure of compression (6, the first number in the list). The fourth number is an optimistic estimate of the number of literals needed in the clause and the first is a measure based on the other three. See the next section and [8] for full details. Progol then adds the best generalisation to its clause base and retracts any clauses which have now been made redundant. Note that 3 clauses are retracted. Usually Progol would now move onto the next example and repeat this procedure. However, all examples are now covered, so Progol stops and shows its constructed definition.

4 The theory

The Progol system uses an approach to the general problem of ILP called mode directed inverse entailment (MDIE). In contrast to inverse resolution [10] and subsumption oriented approaches to induction [13], inverse entailment is based upon model-theory rather than resolution proof-theory. In this way a great deal of clarity and simplicity can be achieved. Furthermore by basing induction on a sounder theoretic footing, it is hoped that it is easier to develop completeness and consistency results. MDIE is a generalisation and enhancement of these previous approaches.

4.1 The general problem in ILP

The general problem in ILP can be summarized as follows. Given background knowledge B and examples E find the simplest consistent hypothesis H such that

$$B \wedge H \models E$$

If we rearrange the above using the law of contraposition we get the more suitable form

$$B \wedge \overline{E} \models \overline{H}$$

In general B, H and E can be arbitrary logic programs but if we restrict H and E to being single Horn clauses, $\overline{H}$ and $\overline{E}$ above will be ground skolemised unit clauses. If $\overline{\perp}$ is the conjunction of ground literals which are true in all models of $B \wedge \overline{E}$ we have

$$B \wedge \overline{E} \models \overline{\perp}$$

Since $\overline{H}$ must be true in every model of $B \wedge \overline{E}$ it must contain a subset of the ground literals in $\overline{\perp}$. Hence

$$B \wedge \overline{E} \models \overline{\perp} \models \overline{H}$$

and so

$$H \models \perp$$

A subset of the solutions for H can then be found by considering those clauses which θ-subsume $\perp$. The complete set of candidates for H could in theory be found from those clauses which imply $\perp$. As yet Progol does not attempt to find a fuller set of candidates (bypassing the undecidabilty of implication between clauses with bounds on the number of resolution steps in the Prolog interpreter). Progol searches the latter subset of solutions for H that theta-subsume $\perp$.

4.2 Mode declarations

In general $\perp$ can have infinite cardinality. Progol uses the head and body mode declarations together with other settings to build the most specific clause and hence to constrain the search for suitable hypotheses.

A mode declaration has either the form *modeh(n, atom)* or *modeb(n, atom)*. where n, the recall, is an integer greater than zero or '*' and atom is a ground atom. Terms in the atom are either normal or place-marker. A normal term is either a constant or function symbol followed by a bracketed tuple of terms. A place-marker is either +type, -type or #type where type is a constant.

The recall is used to bound the number of alternative solutions for instantiating the atom. A recall of '*' indicates all solutions - in practice a large number. +type, -type, #type correspond to input variables, output variables and constants respectively.

Progol imposes a restriction upon the placement of input variables in hypothesised clauses. Suppose the clause is written as $h \ :- \ b_1, \ldots, b_n$ where h is the head atom and $b_i, 1 \leq i \leq n$ are the body atoms. Then every variable of +type in any atom b_i is either of +type in h or -type in some atom b_j where $1 \leq j < i$. This imposes a quasi-order on the body atoms and ensures that the clause is logically consistent in its use of input and output variables.

4.3 An example

In order to illustrate how Progol finds consistent hypotheses, the following example will be used (this is similar to the grammar example file in the Progol4.4 release).

```
%%%%%%%%%%%%%%%%%%%%
% Grammar learning problem. Learns a simple English language
% phrase grammar.

% Increase to 100 the resolution bound of the Prolog interpreter,
% ie. the maximum number of unifications the interpreter will
% perform per call.
:- set(r,100)?

% Increase to 1000 the depth bound for the Prolog interpreter.
:- set(h,1000)?

% Set learning from positive examples only.
:- set(posonly)?
```

```
% Learn grammar rules with head s(In,Out) and body atoms representing
% determiners, prepositions, nouns, etc.
:- modeh(1,s(+wlist,-wlist))?
:- modeb(1,det(+wlist,-wlist))?
:- modeb(1,prep(+wlist,-wlist))?
:- modeb(1,noun(+wlist,-wlist))?
:- modeb(1,tverb(+wlist,-wlist))?
:- modeb(1,iverb(+wlist,-wlist))?
:- modeb(*,np(+wlist,-wlist))?
:- modeb(*,vp(+wlist,-wlist))?

%%%%%%%%%%%%%%%%%%%%
% Types

wlist([]).
wlist([W|Ws]) :- word(W), wlist(Ws).

word(a).  word(at).  word(ball). word(big).  word(dog). word(every).
word(happy).  word(hits).  word(house).  word(in).  word(man). word(nice).
word(on). word(small).  word(takes).  word(the).  word(to).  word(walks).

%%%%%%%%%%%%%%%%%%%%%%%
% Background knowledge

% The following represents the grammar rule
% NP -> DET NOUN
np(S1,S2) :- det(S1,S3), noun(S3,S2).
np(S1,S2) :- det(S1,S3), adj(S3,S4), noun(S4,S2).

% The following represents the grammar rule
% DET -> a
det([a|S],S).
det([the|S],S).
det([every|S],S).

vp(S1,S2) :- tverb(S1,S2).
vp(S1,S2) :- tverb(S1,S3), prep(S3,S2).

noun([man|S],S).
noun([dog|S],S).
noun([house|S],S).
noun([ball|S],S).

% Transitive and intransitive verbs.
```

```
tverb([hits|S],S).
tverb([takes|S],S).
tverb([walks|S],S).

iverb([barks|S],S).
iverb([hits|S],S).
iverb([takes|S],S).
iverb([walks|S],S).

prep([at|S],S).
prep([to|S],S).
prep([on|S],S).
prep([in|S],S).
prep([from|S],S).

adj([big|S],S).
adj([small|S],S).
adj([nice|S],S).
adj([happy|S],S).

%%%%%%%%%%%%%%%%%%%%
% Positive examples

s([the,man,walks,the,dog],[]).
s([the,dog,walks,to,the,man],[]).
s([a,dog,hits,a,ball],[]).
s([the,man,walks,in,the,house],[]).
s([the,man,hits,the,dog],[]).
s([a,ball,hits,the,dog],[]).
s([the,man,walks],[]).
s([a,ball,hits],[]).
s([every,ball,hits],[]).
s([every,dog,walks],[]).
s([every,man,walks],[]).
s([a,man,walks],[]).
s([a,small,man,walks],[]).
s([every,nice,dog,barks],[]).
```

In the above, bounds on the on the dimensions of proofs carried out by the Prolog interpreter inside Progol, are set first. The resolution bound r (maximum number of unifications) is increased from its default value of 400 [1] to 1000 and the depth bound (maximum stack depth before forced backtracking) from its default of 30 to 100. Whenever the interpreter reaches a limit it gives a warning message and

[1] Progol's default settings can be found using the *settings* command when running Progol interactively.

returns failure so the two sets of commands are just a precaution to avoid this happening.

The aim of the example is to find grammar rules which parse simple English sentences such as "The man walks the dog". The mode declarations express this and also define the input(+) and output(-) variables and the types of the parameters - in this case they all have the same type *wlist* (meaning word list). We then define word lists recursively in terms of a given set of words which appear in the examples and background knowledge. Finally 14 positive examples of simple English sentences are given. In practice this is a very small number but is sufficient for learning a few simple grammar rules.

4.4 Construction of the most specific clause

The construction of the most specific clause $\perp$ proceeds as follows.

The first clause e to generalise is

```
s([the,man,walks,the,dog],[]).
```

From the head mode declaration

```
:- modeh(1,s(+wlist,-wlist))?
```

we have the trivial deduction

$$B \wedge \overline{e} \models \overline{s([the, man, walks, the, dog], [])}$$

From the body mode declaration

```
:- modeb(1,det(+wlist,-wlist))?
```

and replacing the input variable by [the,man,walks,the,dog] we have the deduction

$$B \wedge \overline{e} \models det([the, man, walks, the, dog], [man, walks, the, dog])$$

Note that this constructs the term [man,walks,the,dog] in place of the output variable. Using this new term with the body mode declaration

```
:- modeb(1,noun(+wlist,-wlist))?
```

and replacing the input variable by the new term we have the deduction

$$B \wedge \overline{e} \models noun([man, walks, the, dog], [walks, the, dog])$$

However, using other mode declarations in a similar way we can get the following deductions as well.

$$\begin{aligned}
B \wedge \overline{e} &\models np([man, walks, the, dog], [walks, the, dog]) \\
B \wedge \overline{e} &\models verb([walks, the, dog], [the, dog]) \\
B \wedge \overline{e} &\models vp([walks, the, dog], [the, dog]) \\
B \wedge \overline{e} &\models np([the, dog], [])
\end{aligned}$$

Putting these and all other similar deductions together we get

$$
\begin{aligned}
B \wedge \overline{e} \models \; & \overline{s([the, man, walks, the, dog], [])} \wedge \\
& det([the, man, walks, the, dog], [man, walks, the, dog]) \wedge \ldots \\
& \wedge np([the, dog], [])
\end{aligned}
$$

$\overline{\perp}$ is the right hand side of the above deduction. In effect a restricted minimal Herbrand model has been constructed for $B \wedge \overline{E}$, the mode declarations being used to guide the inclusion of predicates that might be of importance. To derive $\perp$, the above is first negated to give

$$
\begin{aligned}
& s([the, man, walks, the, dog], []) \vee \\
& \overline{det([the, man, walks, the, dog], [man, walks, the, dog])} \vee \ldots \vee \\
& \overline{np([the, dog], [])}
\end{aligned}
$$

and then the most specific clause can be constructed by replacing terms in the above by unique variables.

$$
\begin{aligned}
\perp = \; & s(A, B) \vee \overline{det(A, C)} \vee \overline{np(A, D)} \vee \overline{noun(C, D)} \vee \overline{tverb(D, E)} \vee \\
& \overline{iverb(D, E)} \vee \overline{vp(D, E)} \vee \overline{det(E, F)} \vee \overline{np(E, B)}
\end{aligned}
$$

ie $\perp$ is given by

```
s(A,B) :- det(A,C), np(A,D), noun(C,D), tverb(D,E), iverb(D,E), \\
          vp(D,E), det(E,F), np(E,B).
```

in Prolog notation (see Section 8 to see how to trace the construction of this clause).

4.5 Algorithm for constructing the most specific clause

The general algorithm for constructing the most specific clause is given in Figure 1. The final set $\perp$ is a disjunction of its members so the final clause will have the h constructed in step 2 as its head and the bs constructed in step 4 as atoms in its body.

Note that InTerms is not affected by variables corresponding to -type in step 3 because of the restriction mentioned earlier that an input variable in a body atom must either be an input variable in the head or an output variable in an earlier body atom.

The recall is used to determine how many times to call the Prolog interpreter for each instantiation of the clause in step 4. It may well be that the clause will succeed many times and produce many answer substitutions.

The maximum variable depth (default 3) determines how many times step 4 is executed. The Prolog interpreter inside Progol is also bounded in its number of resolution steps and in its depth.

Let e be the clause $a\ :-\ b_1, \ldots, b_n$.
Then $\overline{e}$ is $\overline{a} \wedge b_1 \wedge \ldots \wedge b_n$.
$hash : Terms \rightarrow N$ is a function uniquely mapping terms to natural numbers.

1. Add $\overline{e}$ to the background knowledge
2. $InTerms = \emptyset$, $\perp = \emptyset$
3. Find the first head mode declaration h such that h subsumes a with substitution θ
   For each v/t in θ,
      if v corresponds to a #type, replace v in h by t
      if v corresponds to a +type or -type, replace v in h by v_k
         where v_k is the variable such that $k = hash(t)$
      If v corresponds to a +type, add t to the set InTerms.
   Add h to $\perp$.
4. For each body mode declaration b
      For every possible substitution θ of variables corresponding to +type
            by terms from the set InTerms
         Repeat recall times
            If Prolog succeeds on goal b with answer substitution θ'
               For each v/t in θ and θ'
                  If v corresponds to #type, replace v in b by t
                  otherwise replace v in b by v_k where $k = hash(t)$
                  If v corresponds to a -type, add t to the set InTerms
               Add $\overline{b}$ to $\perp$
5. Increment the variable depth
6. Goto step 4 if the maximum variable depth has been achieved.

Fig. 1. Algorithm for constructing $\perp$

4.6 Searching the subsumption lattice

Thus, for each example, the search for suitable hypotheses is limited to the bounded sub-lattice

$$\Box \preceq H \preceq \perp$$

where $\preceq$ denotes θ-subsumption and $\Box$ is the empty (false) clause. The sub-lattice has a most general element top which is $\Box$ and a most specific or least general element bottom which is $\perp$. The construction of the most specific clause $\perp$ reduces the search space significantly.

When generalising an example E relative to background knowledge B, Progol constructs $\perp$ and searches from general to specific through the sub-lattice of single clause hypotheses H such that $\Box \preceq H \preceq \perp$. The search space is thus bounded both above and below. The search is therefore better constrained than in other approaches to general to specific searching. For the purpose of searching this lattice of clauses ordered by θ-subsumption, Progol employs a refinement operator. Since $H \preceq \perp$, it must be the case that there exists a substitution θ such that $H\theta \subseteq \perp$. Thus for each literal l in H, there exists a literal l' in $\perp$ such that $l\theta = l'$. The Progol refinement operator has simply to keep track of θ and a list of those literals l' in $\perp$ which have a corresponding literal l in H.

Any clause H which subsumes $\perp$ corresponds to a subset of the literals in $\perp$ with substitutions applied. In Progol we have the added provision that the head atom must always be included in order to obtain a sensible generalisation.

For the most specific clause in the example above, Progol starts from the empty clause

```
false :- true.
```

and first refines it to

```
s(A,B).
```

This clause is then tested to see how many of the positive and negative examples it predicts. The refinements of this clause are then considered. Progol now refines this clause by adding the first body atom det(A,C) to the clause.

```
s(A,B) :- det(A,C).
```

The new clause is then tested against the examples and possible refinements are again considered. Progol is searching a tree whose root node is $\Box$. Children of a node are the possible refinements that can be made to the clause at that node. Certain nodes will be pruned by the use of the user-defined prune statements which delimit the hypothesis space. In this example none of the subsumed clauses are accepted so Progol then backtracks to try

```
s(A,B) :- np(A,C).
```

In this example Progol has to search 21 nodes before finding the most compressive hypothesis

```
s(A,B) :- np(A,C), vp(C,D), np(D,B).
```

Note also that the refinement operator considers only variable/variable substitutions which map hypothesised clauses to subsets of $\perp$. General subsumption can be achieved when necessary by the the use of the equality operator in modeb declarations. See [8] for further details.

4.7 The search algorithm

To search the subsumption lattice Progol applies an A*-like algorithm to find the clause with maximal compression. A simple outline of this algorithm is given in Figure 2.

The algorithm calculates the following for each candidate clause s

$$p_s = number\ of\ positive\ examples\ correctly\ deducible\ from\ s$$
$$n_s = number\ of\ negative\ examples\ incorrectly\ deducible\ from\ s$$
$$c_s = length\ of\ clause\ s - 1$$
$$h_s = number\ of\ further\ atoms\ to\ complete\ the\ clause$$
$$f_s = p_s - (n_s + c_s + h_s)$$

h_s is calculated by inspecting the output variables in the clause and determining whether they have been defined. For example

Algorithm for searching $\Box \preceq C \preceq \bot$

e is the example being generalised

1. $Open = \{\Box\}$, $Closed = \emptyset$
2. $s = best(Open)$, $Open = Open - \{s\}$, $Closed = Closed \cup \{s\}$
3. if $prune(s)$ goto 5
4. $Open = (Open \cup refinements(s)) - Closed$
5. if $terminated(Closed, Open)$ return $best(Closed)$
6. if $Open = \emptyset$ return e (no generalisation)
7. goto 2

Fig. 2. Algorithm for searching the lattice

```
s(A,B).
```

would have $h_s = 3$ because it requires at least three literals from $\bot$ to construct a chain of atoms connecting A to B. This is found from a static analysis of $\bot$.

f_s is a measure of how well a clause s explains all the examples with preference given to shorter clauses.

The function $best(S)$ returns a clause $s \in S$ which has the highest f value in S.

$prune(s)$ is true if $n_s = 0$ and $f_s > 0$. In this case it is not worth considering refinements of s as they cannot possibly do better since any refinement will add another atom to the body of the clause and so cannot have a higher value of p than s does. It also cannot improve upon n_s as the latter is zero.

$terminated(S,T)$ is true if $s = best(S)$, $n_s = 0$, $f_s > 0$ and for each t in T it is the case that $f_s \geq f_t$. In other words none of the remaining clauses nor any potential refinements of them can possibly produce a better outcome than the current one.

This algorithm is guaranteed to terminate and to return the clause (if it exists) which has both maximum explanatory power and high compression as measured by its length.

In the worst case the algorithm will consider all clauses in the subsumption ordering. For further details on bounds for the algorithm and a more theoretical discussion of the Progol system see [8].

4.8 The covering algorithm

Progol uses a simple set cover algorithm to deal with multiple examples. It repeatedly generalises examples in the order found in the Progol source file and

B is the background knowledge, E the set of examples.

1. If $E = \emptyset$ return B
2. Let e be the first example in E
3. Construct clause $\perp$ for e
4. Construct clause H from $\perp$
5. Let $B = B \cup H$
6. Let $E' = \{e : e \in E \textit{ and } B \models e\}$
7. Let $E = E - E'$
8. Goto 1

Fig. 3. Covering algorithm

adds the generalisation to the background knowledge. Examples which are now redundant relative to the new background knowledge are then removed. This is shown in Figure 3.

For the grammar learning problem, only one pass through this algorithm is needed to cover all the examples. This results in the single hypothesised general clause solution.

4.9 The output

The output from Progol when processing the example follows.

```
CProgol Version 4.4

[:- set(r,100)? - Time taken 0.00s]
[:- set(h,1000)? - Time taken 0.00s]
[Noise has been set to 100%]
[Example inflation has been set to 400%]
[The posonly flag has been turned ON]
[:- set(posonly)? - Time taken 0.00s]
[:- modeh(1,s(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(1,det(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(1,prep(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(1,noun(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(1,tverb(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(1,iverb(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(100,np(+wlist,-wlist))? - Time taken 0.00s]
[:- modeb(100,vp(+wlist,-wlist))? - Time taken 0.00s]
[Testing for contradictions]
[No contradictions found]
[Generalising s([the,man,walks,the,dog],[]).]
[Most specific clause is]

s(A,B) :- det(A,C), np(A,D), noun(C,D), tverb(D,E), iverb(D,E),
```

```
vp(D,E), det(E,F), np(E,B).

[Learning s/2 from positive examples]
[C:-3,56,55,3 s(A,B).]
[C:42,56,9,3 s(A,B) :- det(A,C).]
[C:50,56,2,2 s(A,B) :- np(A,C).]
[C:51,52,1,1 s(A,B) :- np(A,C), tverb(C,D).]
[C:50,52,1,1 s(A,B) :- np(A,C), tverb(C,D), iverb(C,D).]
[C:50,52,1,1 s(A,B) :- np(A,C), tverb(C,D), iverb(C,E).]
[C:50,52,1,1 s(A,B) :- np(A,C), tverb(C,D), vp(C,D).]
[C:50,52,1,1 s(A,B) :- np(A,C), tverb(C,D), vp(C,E).]
[C:35,16,1,1 s(A,B) :- np(A,C), tverb(C,D), det(D,E).]
[C:39,16,1,0 s(A,B) :- np(A,C), tverb(C,D), np(D,B).]
[C:35,16,1,1 s(A,B) :- np(A,C), tverb(C,D), np(D,E).]
[C:51,56,1,1 s(A,B) :- np(A,C), iverb(C,D).]
[C:50,52,1,1 s(A,B) :- np(A,C), iverb(C,D), vp(C,D).]
[C:50,52,1,1 s(A,B) :- np(A,C), iverb(C,D), vp(C,E).]
[C:35,16,1,1 s(A,B) :- np(A,C), iverb(C,D), det(D,E).]
[C:39,16,1,0 s(A,B) :- np(A,C), iverb(C,D), np(D,B).]
[C:35,16,1,1 s(A,B) :- np(A,C), iverb(C,D), np(D,E).]
[C:51,52,1,1 s(A,B) :- np(A,C), vp(C,D).]
[C:42,24,1,1 s(A,B) :- np(A,C), vp(C,D), det(D,E).]
[C:44,24,1,0 s(A,B) :- np(A,C), vp(C,D), np(D,B).]
[C:42,24,1,1 s(A,B) :- np(A,C), vp(C,D), np(D,E).]
[21 explored search nodes]
f=44,p=24,n=1,h=0
[Result of search is]

s(A,B) :- np(A,C), vp(C,D), np(D,B).

[6 redundant clauses retracted]
[Generalising s([the,man,walks],[]).]
[Most specific clause is]

s(A,B) :- det(A,C), np(A,D), noun(C,D), tverb(D,B), iverb(D,B),
vp(D,B).

[Learning s/2 from positive examples]
[C:-12,48,55,2 s(A,B).]
[C:34,32,9,2 s(A,B) :- det(A,C).]
[C:46,32,2,1 s(A,B) :- det(A,C), np(A,D).]
[C:40,24,2,1 s(A,B) :- det(A,C), np(A,D), noun(C,D).]
[C:40,24,2,1 s(A,B) :- det(A,C), np(A,D), noun(C,E).]
[C:46,28,1,0 s(A,B) :- det(A,C), np(A,D), tverb(D,B).]
[C:44,28,1,1 s(A,B) :- det(A,C), np(A,D), tverb(D,E).]
```

```
[C:44,28,1,0 s(A,B) :- det(A,C), np(A,D), tverb(D,B), iverb(D,B).]
[C:44,28,1,0 s(A,B) :- det(A,C), np(A,D), tverb(D,B), iverb(D,E).]
[C:44,28,1,0 s(A,B) :- det(A,C), np(A,D), tverb(D,B), vp(D,B).]
[C:44,28,1,0 s(A,B) :- det(A,C), np(A,D), tverb(D,B), vp(D,E).]
[C:47,32,1,0 s(A,B) :- det(A,C), np(A,D), iverb(D,B).]
[C:46,32,1,1 s(A,B) :- det(A,C), np(A,D), iverb(D,E).]
[C:44,28,1,0 s(A,B) :- det(A,C), np(A,D), iverb(D,B), vp(D,B).]
[C:44,28,1,0 s(A,B) :- det(A,C), np(A,D), iverb(D,B), vp(D,E).]
[C:46,28,1,0 s(A,B) :- det(A,C), np(A,D), vp(D,B).]
[C:44,28,1,1 s(A,B) :- det(A,C), np(A,D), vp(D,E).]
[C:42,24,2,1 s(A,B) :- det(A,C), noun(C,D).]
[C:44,24,1,0 s(A,B) :- det(A,C), noun(C,D), tverb(D,B).]
[C:42,24,1,1 s(A,B) :- det(A,C), noun(C,D), tverb(D,E).]
[C:42,24,1,0 s(A,B) :- det(A,C), noun(C,D), tverb(D,B), iverb(D,B).]
[C:42,24,1,0 s(A,B) :- det(A,C), noun(C,D), tverb(D,B), iverb(D,E).]
[C:42,24,1,0 s(A,B) :- det(A,C), noun(C,D), tverb(D,B), vp(D,B).]
[C:42,24,1,0 s(A,B) :- det(A,C), noun(C,D), tverb(D,B), vp(D,E).]
[C:44,24,1,0 s(A,B) :- det(A,C), noun(C,D), iverb(D,B).]
[C:42,24,1,1 s(A,B) :- det(A,C), noun(C,D), iverb(D,E).]
[C:42,24,1,0 s(A,B) :- det(A,C), noun(C,D), iverb(D,B), vp(D,B).]
[C:42,24,1,0 s(A,B) :- det(A,C), noun(C,D), iverb(D,B), vp(D,E).]
[C:44,24,1,0 s(A,B) :- det(A,C), noun(C,D), vp(D,B).]
[C:42,24,1,1 s(A,B) :- det(A,C), noun(C,D), vp(D,E).]
[C:50,48,2,1 s(A,B) :- np(A,C).]
[C:48,28,1,0 s(A,B) :- np(A,C), tverb(C,B).]
[C:46,28,1,1 s(A,B) :- np(A,C), tverb(C,D).]
[C:46,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,B).]
[C:46,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,D).]
[C:46,28,1,0 s(A,B) :- np(A,C), tverb(C,B), vp(C,B).]
[C:46,28,1,0 s(A,B) :- np(A,C), tverb(C,B), vp(C,D).]
[C:44,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,B), vp(C,B).]
[C:44,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,B), vp(C,D).]
[C:44,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,D), vp(C,B).]
[C:44,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,D), vp(C,D).]
[C:44,28,1,0 s(A,B) :- np(A,C), tverb(C,B), iverb(C,D), vp(C,E).]
[C:49,32,1,0 s(A,B) :- np(A,C), iverb(C,B).]
[C:47,32,1,1 s(A,B) :- np(A,C), iverb(C,D).]
[C:46,28,1,0 s(A,B) :- np(A,C), iverb(C,B), vp(C,B).]
[C:46,28,1,0 s(A,B) :- np(A,C), iverb(C,B), vp(C,D).]
[C:48,28,1,0 s(A,B) :- np(A,C), vp(C,B).]
[C:50,44,1,1 s(A,B) :- np(A,C), vp(C,D).]
[48 explored search nodes]
f=49,p=32,n=1,h=0
[Result of search is]
```

```
s(A,B) :- np(A,C), iverb(C,B).

[8 redundant clauses retracted]

s(A,B) :- np(A,C), vp(C,D), np(D,B).
s(A,B) :- np(A,C), iverb(C,B).

[Total number of clauses = 2]

[Time taken 0.32s]
```

The names of the variables may not always match those in the description since Progol generates the name of a variable from its number. The clauses will be equivalent up to alphabetic renaming though.

Once Progol has determined the most specific clause $\perp$, it lists those clauses which subsume it and which might be part of any final hypothesis it reaches. For each clause the 4 integers output correspond to f, p, n and h. These are a measure of predictive power/compression, the number of positive examples explained, the number of negative examples incorrectly explained and the number of further atoms to complete the clause, respectively. Note that f is defined as

$$f = P(p - (n + c + h))/p$$

where c is now the length of the clause and P is the total number of positive examples. The multiplication by P and division by p are just a way of normalising the measure while still leaving f a monotonically increasing function of p. The value of f output is rounded to the nearest integer. Higher values of f indicate the most promising candidate clauses for the final hypothesis.

Also note that the example illustrated is perhaps of one of the most basic kinds that Progol can cope with. It has no constant declarations, functions, lists, integrity constraints, recursion and has few examples and background clauses. It has been used for illustrative purposes only.

5 Estimating accuracy and significance

Once Progol has learnt a set of rules from examples, the system permits a quantifiable analysis of predictive accuracy. The methods used to achieve this vary according to the number of test examples.

Suppose the teaching file `gram_trn.pl` contains the following definitions and examples (the types and background knowledge are assumed the same as in the previous example and have thus been omitted).

```
:- set(r,100)?
:- set(h,1000)?
:- set(posonly)?
```

```
% Learn grammar rules with head s(In,Out) and body atoms representing
% determiners, prepositions, nouns, etc.
:- modeh(1,s(+wlist,-wlist))?
:- modeb(1,det(+wlist,-wlist))?
:- modeb(1,prep(+wlist,-wlist))?
:- modeb(1,noun(+wlist,-wlist))?
:- modeb(1,tverb(+wlist,-wlist))?
:- modeb(1,iverb(+wlist,-wlist))?
:- modeb(*,np(+wlist,-wlist))?
:- modeb(*,vp(+wlist,-wlist))?

%%%%%%%%%%%%%%%%%%%%
% Types

% ...

%%%%%%%%%%%%%%%%%%%%%%%
% Background knowledge

% ...

%%%%%%%%%%%%%%%%%%%%
% Positive examples

s([the,man,walks,the,dog],[]).
s([the,dog,walks,to,the,man],[]).
s([a,dog,hits,a,ball],[]).
s([the,man,walks,in,the,house],[]).

s([the,man,walks],[]).
```

This is almost the example of the last section with a reduction in the positive examples. Given this reduction Progol will fails to find the same hypothesis as before and produces some incorrect classifications.

Once Progol has found a rule for s it can be given a set of test examples to see how accurate its predictions are. Suppose the test file `gram_tst.pl` is as follows.

```
% Positive test examples

s([the,man,hits,the,dog],[]).
s([a,ball,hits,the,dog],[]).
s([a,ball,hits],[]).
s([every,ball,hits],[]).
s([every,dog,walks],[]).
s([every,man,walks],[]).
s([a,man,walks],[]).
```

```
s([a,small,man,walks],[]).
s([every,nice,dog,barks],[]).

% Negative test examples

:- s([every,man],[]).
:- s([a,man],[]).
:- s([a,small,man],[]).
:- s([every,nice,dog],[]).
```

We can call Progol interactively by just typing

```
progol
```

on the command line and then consult the file `gram_trn.pl` to read in the definitions and examples, generalise the predicate in question and then test Progol's hypothesis against the test file. Note that in interactive mode we have to explicitly tell Progol to find a general rule.

```
|- consult(gram_trn)?

|- generalise(s/2)?

s(A,B) :- np(A,C), vp(C,D), np(D,B).
s(A,B) :- np(A,C), tverb(C,B).

|- test(gram_tst)?

[False negative:]s([every,nice,dog,barks],[]).

[PREDICATE s/2]

Contingency table=    ________A________~A
                     P|        8|        0|         8
                      |(     5.5)|(    2.5)|
                    ~P|        1|        4|         5
                      |(     3.5)|(    1.5)|
                       ~~~~~~~~~~~~~~~~~~~
                               9         4         13
[Overall accuracy= 92.31% +/- 7.39%]
[Chi-square = 5.87]
 [Without Yates correction = 9.24]
[Chi-square probability = 0.0154]
```

The *consult* command is used to add the definitions and clauses to Progol's knowledge base. The *generalise* command then tells Progol to find a general rule for *s*. The argument to this command has the form *predicate/arity*. The second rule that Progol finds incorrectly has *tverb* instead of *iverb*. This rule is then tested using the test command which expects a filename as its sole argument. The result of testing are as follows: out of the 9 positive examples in the test all but one were correctly predicted by the rules. The 4 negative examples in the test are all correctly predicted by the rules. The contingency table above shows this - P stands for predicted by the rule, $\sim P$ not predicted, A stands for actual positive examples, $\sim A$ for negative ones. A statistical chi-square test has been applied to the data and an overall accuracy and chi-square probability calculated.

To improve accuracy, one could then try adding the test examples to the learning examples and see if the system can come up with an improved rule. In this case the test rules are a little more general and Progol does come up with a new rule that explains all the examples.

A further testing technique provided by the system is the leave-one-out method. When there are not many examples to learn from and test, an overall accuracy measure can be calculated by putting one example aside for testing and learning a rule from the remaining examples. This is then repeated for all the examples. The procedure to do this is to consult the relevant files and then to call the built-in procedure *leave* with an argument of the form *predicate/arity*. Note that generalise must not be used; otherwise the rule is learnt from all the examples.

Another technique that can be used is that of layered learning. In this case Progol is first allowed to form a general rule and then it is given a file of examples from which it tests and tries to improve its rule. A number of the examples are tested against the general rule and when the first false positive or negative is encountered Progol will improve its rule to take account of this exception. The process is then repeated but now the exceptions are accumulated until at least 5 clauses have been tested before Progol improves its rule from them. The algorithm then repeats with 25, 125, ... clauses to be tested before trying to improve the rule. This command is issued as follows.

```
|- layer(filename)?
```

6 Declarative bias

In the following two sub-sections we describe the main mechanisms in Progol that can be used to control the search[2]. These are a) integrity constraints and b) prune statements. Their effects are illustrated in Figure 4. Note that with integrity constraints if the hypothesis H is disallowed then so are all more general clauses. Conversely, with prune statements, if H is pruned, then so are all more specific clauses in the search. Good use of these declarative bias mechanisms (especially prune statements) can be extremely effective for reducing the amount of search performed by Progol.

[2] Section 7 describes how to set resource bounds which can also be used to control the search.

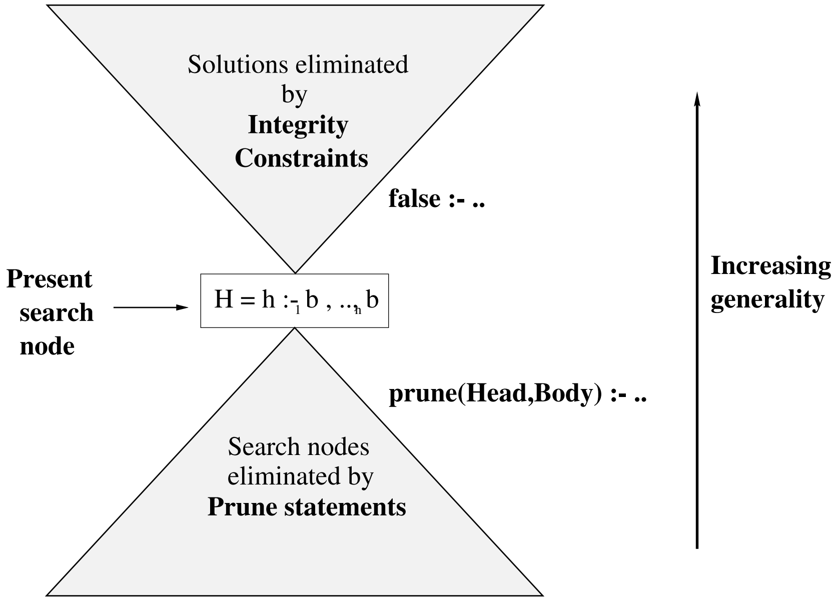

Fig. 4. Controlling search using integrity constraints and prune statements

6.1 Integrity constraints

Progol allows the user to add integrity constraints into the clause base. These are represented as headless clauses, though are stored internally as clauses with head *false*. For example, if we were concerned with the predicates *person*, *male* and *female*, we might include the following clauses.

```
:- person(X), male(X), female(X).
```

This headless clause is stored internally as

```
false :- person(X), male(X), female(X).
```

and can be read as saying "false is provable if there is a person who is both male and female", ie. no person can be both male and female. If the Prolog query *false?* succeeds it means that the integrity constraints have been violated. The integrity constraints can be listed in interactive mode using the query *listing(false)?*. Note that negative examples are just a special case of integrity constraints. If Progol detects that a hypothesis under consideration is inconsistent with this integrity constraint then it will disallow its acceptance.

Closed world integrity constraint

A more advanced example of an integrity constraint is as follows.

```
:- hypothesis(male(X),Body,_), person(X), Body,
   not(clause(male(X),true)).
```

The Progol built-in predicate *hypothesis* returns the present hypothesis under consideration if one exists and fails otherwise. The three arguments of hypothesis are 1) hypothesis head, 2) hypothesis body and 3) the unique clause number of the hypothesised clause. The integrity constraint can be read as saying that "false is provable (ie. a contradiction has been found) if a) the present hypothesis is male(X) :- Body and b) X is a person and c) Body is provable and d) male(X) is not a unit clause (positive example) in the clause base". Such an integrity constraint is often called a "Closed World Assumption", meaning that the hypotheses must not make predictions about examples which have not been seen. The use of such a closed world assumption ensures that Progol will not overgeneralise, which is useful for instance in the case in which no negative examples are available. However, the integrity constraint eliminates overgeneralisation by ensuring that Progol will not generalise at all with respect to the positive examples!

Output completeness integrity constraint

The following is a less severe integrity constraint, which has been referred to as "input completeness" [7]. The constraint requires that the hypothesised predicate must act as functions (ie. $f : X \to Y$ where there is at most one $f(x) \in Y$ for every $x \in X$) with respect to the training examples. In [7] this type of constraint is used for learning rules for the construction of the past tense of English. Thus for every verb it is assumed that there is a unique past tense. The following Progol integrity constraint encodes this constraint.

```
:- hypothesis(past(X,Y),Body,_),
        clause(past(X,Z),true), Body, not(Y==Z).
```

The constraint can be read as saying "false if a) the hypothesis is past(X,Y) :- Body, and b) there is a unit clause (ie. example) past(X,Z) and c) when the Body is proved the substitution for Y differs from Z". In other words the hypothesised clause must not make a prediction for the past tense of a verb in the example set which differs from the actual past tense of that verb.

Generative integrity constraint

The examples above illustrate the use of integrity constraints for testing semantic properties of hypotheses. It is also quite usual to use them to test syntactic properties of clauses. One such widely used syntactic property of clauses is whether they are *generative* [11], otherwise called *range-restricted.* A generative clause is one in which each variable appears in at least two different atoms. It can be shown that if all clauses in a logic program are range restricted then all the atoms which can be derived from it are ground. The following integrity constraint and associated definitions encode the requirement of generativeness for hypothesised clauses.

```
:- hypothesis(Head,Body,_), nongenerative((Head,Body)).

nongenerative(C) :-     % C is a comma-separated list of atoms
        in(A1,C), var(V1,A1),
        not((in(A2,C), A1\==A2, var(V2,A2), U1==V2)).

var(V,V) :- var(V).
var(V,T) :- arg(N,T,T1), N>0, var(V,T1).
```

The built-in predicate *in* tests the membership of its first argument within the comma separated list represented by its second argument. The predicate *nongenerative* succeeds if there is a variable *V1* in atom *A1* of clause *C* where *V1* does not appear in any other atom *A2* in *C*.

6.2 Prune statements

Integrity constraints reject certain hypotheses, but only after the Progol interpreter has constructed the hypothesis to be tested. If pruning is used instead, clauses can be taken out of consideration with little execution time overhead. Consider for example the following prune statement.

```
prune(Head,Body) :- Head, not(Body).
```

This statement will reject an hypothesis generated by Progol if it is found that with a certain substitution the Head can be proved but the Body cannot. Put another way, the prune rule says that for every instance of the head the body must be provable. This means that the target rule set consists of a single clause which covers all of the examples. Thus any hypothesised clause which does not do so can be ignored, as can all more specific clauses. In this way the entire search tree below such an hypothesis can be pruned.

Prune statements are extremely useful for stating which kinds of clause should not be considered in the search. For instance, suppose you wanted to disallow self-recursive clauses. This can be achieved using the following simple prune rule.

```
prune(Head,Body) :- in(Head,Body).
```

The prune statement eliminates any clause whose Head unifies with an atom in its Body. Similarly a single level of mutual recursion can be eliminated using the following prune statement.

```
prune(Head,Body) :-
        in(Atom,Body), clause(Atom,Body1),
        in(Head,Body1).
```

7 Setting resource bounds

It is necessary to put finite limits on a variety of time and space resources used within Progol's search in order to ensure that it terminates efficiently. This is done largely using internal settings of various named bounds. These bounds all have default values which can be viewed using the built-in command *settings?* and can be altered using the built-in commands *set* and *unset*.

Below is a list of the named resource bounds followed by their default values in brackets and a short description of each.

h (Default=30). This is the maximum depth of any proof carried out by the interpreter, ie. the maximum stack depth before backtracking occurs. When the limit is exceeded a warning of the form *[WARNING: depth-bound failure - use set(h,..)]* is issued. This is not an error. It is necessary to have such failures when learning recursive rules.

r (Default=400). This is the maximum depth of resolutions (unifications) allowed in any proof carried out by the interpreter. Once this limit is exceeded the entire proof is failed (unlike *h* in which backtracking occurs when the bound is reached). and the warning *[WARNING: depth-bound failure - use set(r,..)]* is issued. Again this is not an error. It is necessary to have such failures when learning recursive rules.

nodes (Default=200). This is a bound on the number of search nodes expanded in the search of the lattice of clauses. According to sample complexity results in the paper [9] this value should be set to around 1.6 times the number of examples in the training set in order to minimise expected error.

c (Default=4). This is the maximum number of atoms in the body of any hypothesised clause. Increasing this value potentially increases the maximum clause search size exponentially.

i (Default=3). This is a bound on the number of iterations carried out in the construction of the bottom clause. Each iteration introduces new variables based on the instantiation of output terms associated with modeb declarations.

8 Debugging Progol input files

Note that writing background knowledge, integrity constraints and prune statements involves programming. It should by now be clear that each of these components of a Progol input file is itself a piece of Prolog code. Thus it is important to have debugging tools which allow the user to test and correct this code. The available debugging tools are essentially the same as those found in a standard Prolog interpreter (ie. *trace* and *spy*) [1], though there is an additional mechanism in CProgol4.4 to test how a particular bottom clause is being constructed. This mechanism can be illustrated with reference to the worked example from Section 4.4. Suppose that we have loaded the file `grammar.pl` and wish to trace the construction of the most specific clause associated with the example sentence "The man walks the dog". We proceed as follows.

```
|- trace?
|- s([the,man,walks,the,dog],[])!
[Testing for contradictions]
(0) Call: false
(0) Fail: false
[No contradictions found]
```

Note the use of '!' after the example rather than '?'. This tells Progol to construct the most specific clause for this example. The first thing which Progol does is to test if the given example contradicts any integrity constraint. This is done by testing the goal *false?*, which fails, leading to the diagnosis `[No contradictions found]`. Next the *modeh* declaration is matched as follows.

```
(0) Call: s(_0,_1)=s([the,man,walks,the,dog],[])
(0) Done: s([the,man,walks,the,dog],[])=s([the,man,walks,the,dog],[])
```

The first and second arguments are found to be the terms *[the,man,walks,the,dog]* and *[]*. Next these terms are type checked as follows.

```
(0) Call: wlist([the,man,walks,the,dog]) s
(0) Done: wlist([the,man,walks,the,dog])
```

The user's response of 's' causes the tracing of the sub-proof to be skipped, leading to the result that *[the,man,walks,the,dog]* succeeds as a *wlist*. Having determined that the new term is of type *wlist*, Progol then uses the modeb declarations to test this term with respect to the background knowledge as follows.

```
(0) Call: det([the,man,walks,the,dog],_1) s
(0) Done: det([the,man,walks,the,dog],[man,walks,the,dog])
```

Here it is shown that *the* is a determiner followed by the phrase *[man,walks,the,dog]*, which in term becomes a new term which in turn will be tested using the modeb declarations and associated background predicate definitions.

As is usual with Prolog interpreters, tracing can be turned off by responding with *n*, and it is possible to leap to the next spy-point by responding with *l*. Spy-points are set on individual predicate definitions using the *spy* command. Thus turning off tracing using *notrace?* and calling *spy(det/2)?* above would ensure that the trace only started when the arity 2 predicate *det* was called. All spy points can be turned off using *nospy?*.

Notice that a command such as *s([the,man,walks,the,dog],[])!* above only leads to the construction of the bottom clause. If you want to trace the entire search through the lattice, it is necessary to first issue the command *set(searching)?*.

9 Summary

The design methodology for Progol was to present the user with a standard Prolog interpreter augmented with inductive capabilities. The syntax for examples, background knowledge and hypotheses is the Dec-10 Prolog syntax with the usual augmentable set of prefix, postfix and infix operators. Headless Horn clauses are used to represent negative examples and integrity constraints. Indeed it is possible for Progol to learn headless constraints from headless ground unit clauses by the use of a modeh declaration for the predicate *false*. The standard library of primitive predicates described in Clocksin and Mellish [1] is built into Progol and available as background knowledge. Progol constructs new clauses by generalising from the examples in the Prolog database using an inverse entailment algorithm. Results from the theory of ILP guarantee that Progol conducts an admissible search through the space of generalisations, finding the maximally compressive set of clauses from which all the examples can be inferred. The choice of engineering a complete Prolog interpreter was taken in order to make induction a first-class and efficient operation on the same footing as deductive theorem proving. Progol has been used successfully in experiments in learning to predict mutagenic molecules [5, 14], in drug design [4, 3], and in protein shape prediction [12, 15].

Acknowledgements

The first author would like to thank his wife, Thirza Castello-Cortes, and daughter Clare, who have provided unfailing support during the writing of this chapter. This work was supported partly by the Esprit Long Term Research Action ILP II (project 20237), EPSRC grant GR/K57985 on Experiments with Distribution-based Machine Learning and an EPSRC Advanced Research Fellowship held by the author.

The second author would like to thank his wife, Valmai Alysia, for much valued help in proof-reading various versions of this chapter.

References

1. W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.
2. James Cussens, David Page, Stephen Muggleton, and Ashwin Srinivasan. Using Inductive Logic Programming for Natural Logic Processing. In W. Daelemans, T. Weijters, and A. van der Bosch, editors, *ECML'97 – Workshop Notes on Empirical Learning of Natural Language Tasks*, pages 25–34, Prague, 1997. University of Economics. Invited keynote paper.
3. P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacophore discovery using the inductive logic programming system Progol. *Machine Learning*, 30:241–271, 1998.
4. R. King, S. Muggleton, R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, 89(23):11322–11326, 1992.
5. R. King, S. Muggleton, A. Srinivasan, and M. Sternberg. Structure-activity relationships derived by machine learning: the use of atoms and their bond connectives

to predict mutagenicity by inductive logic programming. *Proceedings of the National Academy of Sciences*, 93:438–442, 1996.
6. R.D. King and A. Srinivasan. Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental ealth Perspectives*, 104(5):1031–1040, 1996.
7. R.J. Mooney and M.E. Califf. Induction of first-order decision lists: Results on learning the past tense of english verbs. *Journal of Artificial Intelligence Research*, 3:1–24, 1995.
8. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
9. S. Muggleton. Learning from positive data. *Machine Learning*, 1998. Accepted subject to revision.
10. S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the 5th International Conference on Machine Learning*, pages 339–352. Kaufmann, 1988.
11. S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, London, 1992.
12. S. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.
13. S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.
14. A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.
15. M. Turcotte, S.H. Muggleton, and M.J.E. Sternberg. Protein fold recognition. In C.D. Page, editor, *Proc. of the 8th International Workshop on Inductive Logic Programming (ILP-98)*, LNAI 1446, pages 53–64, Berlin, 1998. Springer-Verlag.