Rapid Prototyping of ILP Systems Using Explicit Bias

William W. Cohen
AT&T Bell Laboratories
Room 2A-427, 600 Mountain Avenue
Murray Hill, NJ 07974
wcohen@research.att.com

Abstract

We describe a system that allows one to rapidly implement a variety of biases for inductive logic programming; this is an especially suitable area for rapid prototyping, as a number of different restrictions on logic programs have been proposed. Our prototyping system is an extension of the Grendel learning system, which accepts two inputs: a set of examples of the concept to be learned, and an explicit description of the desired bias, expressed in a "bias representation language". Rapid prototyping can be accomplished using Grendel by varying the bias description and keeping the rest of the learning system fixed. To allow rapid prototyping for inductive logic programming biases, we have extended Grendel's bias representation language with a powerful macro facility called "lazy macros" that allows succinct descriptions of a number of useful biases. In particular, we present descriptions of the following biases, and compare them experimentally: constant-depth determinate clauses, clauses with a constant number of "free" variables, clauses found via "relational pathfinding", and a previously unexplored bias toward clauses with constant "locality". The last bias is suggested by recent formal results.

1 Introduction

Because learning arbitrary logic programs is difficult, efficient ILP systems have imposed a number of restrictions on the programs they learn. For example, LINUS learns only "constrained" clauses [Džeroski and Lavrac, 1991], GOLEM learns "determinate" clauses of bounded depth [Muggleton and Feng, 1992], and CLINT learns clauses with few free variables [Raedt and Bruynooghe, 1992]. In addition, recent formal results [Džeroski *et al.*, 1992; Cohen, 1993b; Cohen, 1993a] have suggested a number of other potentially useful biases. However, there is as yet no clear understanding of how these biases compare on real-world problems; in fact, some of the biases suggested by formal analysis have not (up until now) been implemented. A rapid prototyping system is an ideal mechanism for exploring this space. In this paper, we will describe such a system.

Our prototyping system is based on a learning system called Grendel [Cohen, 1992]. In Grendel the user is provided with a *language* to express his or her choice of a bias; this language is called the *bias representation language (BRL)*. More specifically, Grendel accepts two inputs: a set of examples of the concept to be learned, and an *explicit description of the hypothesis space to be searched*—that is, a description of the desired *bias* of the learning system—written in the BRL.

This approach has several advantages over more traditional learning systems: for example, it allows finer control over the learning system, and (in the BRL) a means of describing biases to a naive user. Here we will make use of a third advantage of the BRL approach, namely the *modularity and flexibility* of such a learning system. As the learning module itself can be re-used for many different biases, it is often straightforward to implement a new bias, or a variation of an old bias; usually it is possible to vary only the the BRL input, while keeping the core learning system fixed. This is the method we will use in developing our rapid prototyping system.

In this paper, we will first describe Grendel, along with the extensions we have made to its BRL. We will then describe a number of plausible biases, explain how they can be specified in the BRL, and compare them experimentally using Grendel. We will begin with the bias of constant-depth determinate clauses used by GOLEM, and then investigate alternative restrictions on Horn clauses that allow indeterminacy. Most of these alternative restrictions are formally analyzed in a companion paper [Cohen, 1993b]; this paper thus also serves as an experimental verification that these restrictions are practically interesting. In particular, the formal results suggest a previously unexplored bias towards "local" clauses, and experimental results from this paper confirm that this bias is useful.

2 Grendel and its bias representation language

2.1 Antecedent description grammars

The choice of a bias description language is crucial for an explicitly biased learning system; to realize the advantages of controllability, understandability and flexibility, the bias description language must be easy for a user to read, easy to compile into, and easy for the learning system to use. In this section we will describe Grendel's BRL, which is based on context-free grammars.

Many ILP systems generate hypotheses of the form

$$
\begin{array}{l}
\mathrm{p}(\mathrm{X}_1,\ldots,\mathrm{X}_n) \text{ :-body}_1 \\
\mathrm{p}(\mathrm{X}_1,\ldots,\mathrm{X}_n) \text{ :-body}_2 \\
\vdots \\
\mathrm{p}(\mathrm{X}_1,\ldots,\mathrm{X}_n) \text{ :-body}_k
\end{array}
$$

where each clause is function-free, and each body$_i$ is a conjunction of predicates from some predetermined set of "feature predicates". Let us consider how this space of hypotheses can be succinctly represented. As the consequents of the clauses in a hypothesis are the same, let us focus on describing the space of possible clause antecedents.

Notice that the antecedent of each clause is essentially a string of literals. Thus, a compact and general way of representing a set of possible antecedents is with a grammar—specifically, an extended context-free language that generates a string of literals, rather than a string of symbols from a finite alphabet. We call such a grammar an *antecedent description grammar (ADG)*.

An example of an ADG is shown in Figure 1; this grammar implements a bias toward constrained Horn clauses for the well-known *illegal* problem [Quinlan, 1990; Pazzani and Kibler, 1992][1]. The nonterminals of this example grammar have the following meanings: *pred2(X,Y)* expands to some binary predicate relating X and Y; *pred3(X,Y,Z)* is analogous for arity-three predicates; *rel(A,B,C,D,E,F)* expands to some "relationship" between a subset of the variables A, B, C, D, E and F — in particular, either a binary relationship between some pair of these variables, or

[1]We use a notation similar to that used in Prolog for definite clause grammars: terminal symbols of the language are enclosed in square brackets (*e.g*, *[equal(X,Y)]*), commas denote concatenation, an empty pair of square brackets (*e.g.*, []) denotes the empty string, and capital letters like A and X denote logical variables. The semantics of the grammar are as one would expect—it differs from a standard context free grammar only in manipulating logical terms, rather than symbols from a fixed alphabet.

body(illegal(A,B,C,D,E,F)) → rels(A,B,C,D,E,F).

rels(A,B,C,D,E,F) → [].
rels(A,B,C,D,E,F) → rel(A,B,C,D,E,F),rels(A,B,C,D,E,F).

rel(A,B,C,D,E,F) → pred3(A,A,A).
rel(A,B,C,D,E,F) → pred3(A,A,B).
⋮
rel(A,B,C,D,E,F) → pred3(F,F,F).

rel(A,B,C,D,E,F) → pred2(A,A).
rel(A,B,C,D,E,F) → pred2(A,B).
⋮
rel(A,B,C,D,E,F) → pred2(F,F).

pred2(X,Y) → [equal(X,Y)].
pred2(X,Y) → [¬equal(X,Y)].
pred2(X,Y) → [adj(X,Y)].
pred2(X,Y) → [¬adj(X,Y)].

pred3(X,Y,Z) → [between(X,Y,Z)].
pred3(X,Y,Z) → [¬between(X,Y,Z)].

Figure 1: Antecedent description grammar for constrained clauses

an arity-three relationship between some triple; finally, *rels(A,B,C,D,E,F)* expands to a string (possibly empty) of "relationships" between the variables A, B, C, D, E, and F.

The *body(G)* notation used in the first production is special; it is used to denote the start symbol of the grammar for the predicate G. Thus the first production says simply that the antecedent of a clause for *illegal* must be some string of relationships between its six arguments.

An ADG specifies a "hard constraint" on the clauses that the learner can generate; another common type of bias specifies an *order* in which the hypothesis space is to be searched. Grendel uses a very simple mechanism for describing such *preference biases*. Any production of the grammar can be marked as *deferred.* The learner is constrained to generate antecedents using only the non-deferred portion of the grammar, unless the data strongly suggest that this space contains no useful hypotheses;[2] only in this case is the learner allowed to use the full grammar. A rule is marked as deferred by prefixing it with an exclamation point.

For example, marking the following two rules of the grammar of Figure 1 would advise Grendel that clauses with no negated binary literals should be preferred.

!pred2(X,Y) → [¬equal(X,Y)].
!pred2(X,Y) → [¬adj(X,Y)].

In Grendel, ADG's are used to generate clauses. The space of possible clauses defined by an ADG is searched using a top-down, separate-and-conquer greedy search procedure similar to that used by FOIL [Quinlan, 1990]. The search procedure is discussed briefly in [Cohen, 1992] and at length in [Cohen, 1991].

[2] More precisely the non-deferred portion of the grammar is used unless no clauses derived by these rules (and visible to the search procedure) have positive information gain. In addition, Grendel's search procedure imposes a preference bias towards smaller and more general hypotheses. One could, of course, devise much richer languages for expressing preferences; for example, one could devise a declarative language for specifying search control knowledge for the learning module, or augment the grammar so that a Bayseian prior is associated with each generated antecedent. However, so far this simple preference mechanism has proved adequate.

1. While the grammar contains an undefined nonterminal symbol N— *i.e.*, one that does not appear on the left-hand side of any rewrite rule—do the following:

2. Choose some such nonterminal N, and

   (a) find all macros $A_i \rightarrow B_i$ *where* P_i such that A_i and N unify with mgu σ_i

   (b) add to the grammar all rules of the form $A_i\sigma_i\theta_{i_j} \rightarrow B_i\sigma_i\theta_{i_j}$ where θ_{i_1}, θ_{i_2}, ..., are the substitutions associated with the possible proofs of $P_i\sigma_i$.

Figure 2: The algorithm for lazy macro expansion

2.2 Lazy macros

Figure 1 shows that ADG's can be large, even for the highly restricted bias of constrained Horn clauses; notice that the grammar requires $6^2 + 6^3 = 252$ rewrite rules for the nonterminal *rel/6*. To make it convenient for the user to construct such a grammar,[3] the original implementation of Grendel had a simple macro facility. For any Prolog goal P, a rewrite rule of the form $A \rightarrow B$ *where* P was macro-expanded to a series of rules $A\theta_1 \rightarrow B\theta_1$, $A\theta_2 \rightarrow B\theta_2$, ..., where $\theta_1, \theta_2, \ldots$ are the substitutions associated with the possible proofs of P. For instance, the 252 rules for *rel/2* and *rel/3* could have been written with the two macros

rel(A,B,C,D,E,F) → pred3(X,Y,Z) where
    member(X,[A,B,C,D,E,F]),member(Y,[A,B,C,D,E,F]),member(Z,[A,B,C,D,E,F]).
rel(A,B,C,D,E,F) → pred2(X,Y) where
    member(X,[A,B,C,D,E,F]),member(Y,[A,B,C,D,E,F]).

The current implementation of Grendel retains the same syntax for macros, but uses different rules for expanding them: we call the new macros *lazy macros*. Grendel now first reads in all rules and all macros, and then performs the procedure described in Figure 2. To summarize, lazy macros are expanded only when a undefined nonterminal symbol is detected, and are then used only to provide a definition for that specific nonterminal.

One possible use of lazy macros is illustrated by the pair of macros

M1. or(N1,N2) → N1 where true.
M2. or(N1,N2) → N2 where true.

(The "where" keyword indicates to the system that these are macros.) Supposing that one now replaces the four rewrite rules for *pred/2* in the grammar of Figure 1 with the two rules

O1. pred2(X,Y) → or([equal(X,Y)],[¬equal(X,Y)]).
O2. pred2(X,Y) → or([adj(X,Y)],[¬adj(X,Y)]).

Since the nonterminal symbol *or/2* is not defined in the grammar, lazy macro expansion will use macros M1-M2 to construct the following rewrite rules:

[3] Notice that we are not, at least for the moment, concerned with the fact that the grammar is large: given that the space of variable bindings for a predicate is exponential in the arity of that predicate, the grammar *must* be large. Our concern now is with the fact that such large grammars are hard to write down, and more especially, hard to read.

Problem-independent macros:

body(Head) → constrained_lits(Vars) where variables_of(Head,Vars).

constrained_lits(Vars) → [ ] where true.
constrained_lits(Vars) → constrained_lit(Vars),constrained_lits(Vars) where true.

constrained_lit(Vars) → [Lit] where predicate(Lit),variables_of(Lit,LitVars),members(LitVars,Vars).
constrained_lit(Vars) → [¬Lit] where predicate(Lit),variables_of(Lit,LitVars),members(LitVars,Vars).

members(Xs,Ys) :- *each X in the list Xs is a member of the list Ys*
variables_of(Lit,Xs) :- *Xs is a list of the arguments to Lit*

Problem-dependent declarations:

predicate(between(X,Y,Z)). predicate(adj(X,Y)). predicate(equal(X,Y)).

Figure 3: ADG for constrained clauses using lazy macros

X1.1. or([equal(X,Y)],[¬equal(X,Y)]) → [equal(X,Y)].
X1.2. or([equal(X,Y)],[¬equal(X,Y)]) → [¬equal(X,Y)].
X2.1. or([adj(X,Y),[¬adj(X,Y)]) → [adj(X,Y)].
X2.2. or([adj(X,Y)],[¬adj(X,Y)]) → [¬adj(X,Y)].

Taken together, rules O1-O2 and X1.1-X2.2 are equivalent to the original definition of *pred/6*; however, since the user need only write down rules O1 and O2, the new definition is more compact. It should also be noted that lazy macro expansion is *necessary* to correctly implement the *or/2* nonterminal: the original macro expansion strategy would add to the the grammar the two rules *or(N1,N2)→N1* and *or(N1,N2)→N2*, neither of which is even a well-formed rewrite rule (since the right-hand side is not a string of literals.)

This example shows that these "lazy macros" are much more powerful than Grendel's original macro language. Rather than acting as programs for generating rules of the grammar, they act as *methods for completing an incomplete grammar.* Lazy macros are in some sense a "higher order" description of a bias than ADG's; this is illustrated by the macros for *or/2*, which include *variables that are bound to arbitrary symbols of the grammar.* Such variables are not allowed in ordinary ADG's. Lazy macros can be used to substantially extend the syntax of the BRL; for instance, with five additional one-line macros, we can implement the rest of the standard regular expression syntax.

As another example of the "second-order flavor" of lazy macros, consider the ADG in Figure 3, which describes the same bias as Figure 1. Notice that these macros construct a complete grammar from the declarations of the feature predicates; the size of the grammar produced by these macros will be polynomial in the number of arguments to the target predicate and the number of predicates declared, but exponential in the maximal arity j of a declared predicate.

In the next few sections, we will use lazy macros to specify some less restrictive ILP biases.

3 Constant-depth determinate clauses

The GOLEM system [Muggleton and Feng, 1992] learns *ij-determinate* clauses. This bias or variations of it have subsequently been adopted by FOIL [Quinlan, 1991] and LINUS [Lavrač and

Džeroski, 1992]; formally, it is known that a single ij-determinate clause can be pac-learned.[4] In this section we will describe how this bias can be encoded by lazy macros and ADG's.

It is known that for constant i and j the maximum length of an ij-determinate clause is polynomial in the arity of the head of the clause and the number of predicates. We thus begin with the macro

body(Head) → most_of(OutputLits),constrained_lits(Vars) where
  max_depth(I),all_determinate_output_lits(I,Head,OutputLits,Vars).

This macro makes use of a Prolog predicate that takes a depth bound *I* and a literal *Head* that is the head of the clause, and generates a (polynomial-size) list containing all of the determinate literals that have output variables, together with a (again polynomial-size) list *Vars* of all the variables appearing in these literals. Every ij-determinate clause is thus simply a conjunction of "most of" these output literals, followed by some additional literals that contain only variables from the list *Vars*.[5] Finally, the *most_of* nonterminal expands to a string containing as many of the literals in the list *OutputLits* as possible. It can be implemented with three additional macros:

most_of([ ]) → [ ] where true.
most_of([Lit|Lits]) → [Lit],most_of(Lits) where true.
!most_of([Lit|Lits]) → NullLits,most_of(Lits) where
  the output variables of Lit are $[X_1,\ldots,X_k]$*, and*
  NullLits is the string $[X_1$=**null**$,\ldots,X_k$=**null**$]$ *for some null value* **null**.

Prefixing a macro with an exclamation point means that the rules it generates will be *deferred*: recall that the learning system incorporates a preference bias against using the deferred portion of the grammar. These macros thus express a preference bias encouraging the list of output literals to be as long as possible; this preference bias is also used by FOIL, and appears to be necessary for top-down learning of determinate clauses. Again, for fixed i and j the grammar generated by these macros will be of polynomial size.

The left-side side of Table 1 presents results obtained by using Grendel with an ij-determinacy bias[6] on several benchmark problems.[7] (The right-hand side of the table will be explained shortly.) In the table, "m" is the number of training examples, "i" is the depth bound, "error" is the error rate in percent, and "#rules" is the number of rules in the grammar after expansion. "Search" records the number of clauses tested against the data, and is an indirect measure of CPU time; for noise-free problems, such as the ones shown here, CPU time is linear in the search space and the

[4]However, there is no known pac-learning algorithm for a program with polynomially many such clauses; hence it is appropriate to use heuristic techniques (such as those used by Grendel) to learn with this bias.

[5]Although we omit the details, the predicate *all_determinate_output_lits* is straightforward to implement, given declarations that state the "modes" with respect to which each predicate is determinate. Note also that rather than forcing the user to explicitly specify a bound on the arity of literals in clause bodies, we have simply assumed that the feature predicates are all of arity j or less.

[6]For pedagogical reasons, we have presented simplified bias descriptions in the text of the paper. The actual BRL used in the experiments is somewhat more complex, as noted below.

[7]The first set of benchmarks are function-free versions of the programs. The training data sets for *list* and *member* are as described by Quinlan [1990]; for *append* and *reverse*, we used the training data distributed with FOIL, and the same data set restricted to three-element lists; for *multiply* the samples were complete for integers less than four, six, eight and ten, respectively. For these benchmarks, the output of the learner was examined manually, and given an error of "$= 0$" if it was equivalent to the desired program and "> 0" otherwise. The *illegal* problem is as described by Pazzani and Kibler [1992] (using 1000 randomly generated examples) and the *family* problem is as described by Quinlan [1990]. In the *loan* problem, the task is to predict if student loan payments can be deferred, based on enrollment and employment information [Pazzani and Brunk, 1991]. Results on these benchmarks are the average of 20 trials in which m examples were used for training, and the remaining examples for testing.

Table 1: Comparing ij-determinacy and k-free biases

Benchmark	m	ij-determinate bias				k-free bias			
		i	error (%)	search	#rules	k	error (%)	search	#rules
list	6	1	$= 0$	21	14	2	$= 0$	37	26
member	13	1	$= 0$	30	24	2	$= 0$	16	26
append	166	1	$= 0$	1613	227	3	> 0	280	668
	8991		$= 0$	1525			> 0	342	
multiply	64	2	$= 0$	411	121	2	> 0	179	120
	216		$= 0$	655			> 0	295	
	512		$= 0$	774			$= 0$	263	
	1000		$= 0$	931			$= 0$	263	
reverse	100	3	$= 0$	1799	279	4	> 0	248	190
	1156		$= 0$	1798			> 0	426	
illegal	100	0	3.7	281.2	36	0	3.7	281.2	35
						1	8.4	873.4	261
						2	8.5	1128.5	1097
	800	0	0.5	939.1	36	0	0.5	939.1	35
loan	100	0	35.8	36.7	10	0	35.8	33.4	9
		1	33.9	1823.0	34	1	27.2	1538.2	47
						2	14.5	1506.4	299
						4	9.7	2334.6	349
						6	9.7	10344.0	8426
	800	1	36.2	2234.7	34	4	0.6	6620.4	349
family	100	0	63.8	180.0	588	0	63.8	180.0	576
		1	8.3	31137.8	13476	1	0.83	770.2	2316
						2	0.00	1141.5	5761

number of examples.[8] The depth bound i was set to the minimum value necessary for the first five benchmarks, and to the maximum value useful for the remaining benchmarks.[9]

These results show that ij-determinacy is useful on some problems, notably when learning simple Prolog programs like *append*. However, on the *loan* benchmark the bias appears to be overly restrictive;[10] also, for the *family* benchmark, ij-determinacy leads to a surprisingly large search space.

This experiment also showcases the ability of Grendel's BRL to easily incorporate several diverse constraints. The actual BRL used for this experiment contains a number of additional features, including support for negated literals, typed variables, and an interface to a sublanguage for describing symmetries similar to that used in FOCL [Pazzani and Kibler, 1992]. However, it still contains only five macros and 24 lines of code, including the code to generate determinate output literals.

[8] For purposes of comparison, our Prolog implementation searches the 30000+ clauses for *family* with $i = 1$ required about 40 minutes on a Sparc 1+; generating the 13000+ rules took about five minutes.

[9] The *illegal* problem has no determinate relations, so i was set to zero; notice that ij-determinacy with $i = 0$ is equivalent to the bias of constrained clauses. For the *loan* problem, typing constraints mean that increasing the depth bound does not change the search space. On the *family* benchmark depth was limited to $i = 1$ because of the extremely large search space and grammars.

[10] The error on *loan* is high because the target concept appears to require indeterminate predicates.

4 Clauses with k connected free variables

We will next consider some biases that allow indeterminacy. Let us call variables that appear in the body of a clause but not its head *free variables*, and define a variable in a clause to be *connected* if it either appears in the head of the clause, or if it appears in some literal in the body that also contains a connected variable. One reasonable bias is the language of *k-free connected clauses*, defined to be those clauses with at most k free variables, all of which are connected. Formally, the language of k-free clauses is *polynomially predictable* iff DNF is polynomially predictable [Cohen, 1993b; Cohen, 1993a]; since good heuristics for learning DNF exist, it is reasonable to hope that good heuristic learning methods can be found for k-free clauses.

To define this bias with lazy macros, let us define a nonterminal *kfree_lits(ConnVars, Vars)* that will expand to any string of connected literals containing variables from the list *Vars*; we will assume that *ConnVars* is a list of those variables in *Vars* that are already connected. Given this nonterminal, the overall bias could be defined with the macro

```
body(Head) → kfree_lits(ConnVars,Vars) where
     variables_of(Head,ConnVars),append(ConnVars,[Y1,...,Yk],Vars).
```

The nonterminal *kfree_lits* can be defined with two macros. The first one gives the base case: it states that an empty string is a valid string of k-free connected literals

```
kfree_lits(ConnVars,Vars) → [ ] where true.
```

The second macro is a bit more complex:

```
kfree_lits(ConnVars,Vars) → [Lit],kfree(ConnVars1,Vars) where
1          predicate(Lit),
2          variables_of(Lit,LitVars),
3          members(LitVars,Vars),
4          variables_intersect(LitVars,ConnVars),
5          add_to_list(LitVars,ConnVars,ConnVars1).
```

Lines 1-3 of the "where" condition ensure that the new literal *Lit* being added to the string contains only variables from the list *Vars*. Assuming that *ConnVars* is in fact a list of connected variables, line 4 ensures that the literal is "connected" as defined above. Finally, line 5 updates the list *ConnVars* appropriately, saving the information that the variables in *Lit* are now "connected". It can be easily shown that for a fixed maximum predicate arity j and fixed k the macros above will generate only a polynomial number of rules.[11]

The right-hand side of Table 1 gives experimental results on our benchmark problems using a k-free bias.[12] For the program-induction benchmarks, we used for k the number of free variables in the target program, and (as an additional broad hint to the learner) also gave the correct type for the free variables; in spite of this, Grendel with the k-free bias only learned a correct program on a few of these benchmarks. For the other benchmarks, we show results for a range of values for k.[13] Although the k-free bias does poorly on program-induction problems, it does comparatively well on the *family* and *loan* benchmarks, where ij-determinacy is less appropriate.

[11] The grammar is exponential in k, because there will be one nonterminal symbol of the form *kfree(ConnVars, Vars)* for every possible set of connected variables *ConnVars*. Since there are 2^k such subsets, there will be at least 2^k nonterminals. This issue will be discussed again in Section 7.

[12] Again, the results in the table are actually for a more complex k-free bias, containing four macros and 27 lines of code, allowing for negated literals, recursion, typed variables, and symmetry.

[13] Note that a 0-free bias, like a $0j$-determinate bias, is equivalent to a bias of constrained clauses.

Table 2: Comparing alternative biases allowing indeterminate clauses

Bias	k	*loan* (m=100) error(%)	(SE)	search	#rules	*family* (m=100) error(%)	(SE)	search	#rules
constrained		35.8	(0.1)	33.4	9	63.8	(2.0)	180.0	576
determinate		33.9	(0.7)	1823.0	34	8.3	(2.0)	31137.8	13476
k-free	1	27.2	(0.9)	1538.2	47	0.8	(0.6)	770.2	2316
	2	14.5	(0.3)	1506.4	299	0.0	(0.0)	1141.5	5761
	4	9.7	(0.5)	2634.6	349				
k-local	1	33.8	(0.1)	183.6	12	0.8	(0.8)	650.9	1104
	2	3.2	(0.7)	834.4	149	0.0	(0.0)	22857.0	3636
	3	5.0	(0.7)	3657.2	1394				
k-path	1	*not applicable*				63.8	(2.0)	28.0	264
	2					0.0	(0.0)	135.8	3168

5 Clauses of locality k

We will now consider an alternative way of allowing indeterminacy. In evaluating this bias, we will focus on the *family* and *loan* benchmarks, as determinate or constrained logic programs appear to be suitable for the other benchmarks.

If X and Y are two variables appearing in a clause $A{:}{-}B_1 \wedge \ldots \wedge B_r$, let us say that that X *touches* Y if they appear in the same literal. We now define *influences* to be the transitive closure of *touches*, and define the *locale* of a variable X to be the set of literals that contain variables influenced by X; finally, the *locality* of a clause is the size of largest locale of any *free* variable in that clause.[14] As examples, the following clauses have locality one, two and three respectively; free variables of the clauses are highlighted, and the locale of each free variable is underlined.

father(F,S) :- son(S,F) ∧ husband(F,**W**).
no_payment_due(S) :- enlist(S,**PC**)∧peace_corps(**PC**).
draftable(S) :- citizen(S,**C**)∧united_states(**C**) ∧ age(S,**A**)∧(**A**≥18)∧(**A**≤26).

Another possible bias is to consider only k-local clauses—*i.e.*, clauses of locality k or less. This language is incomparable to the language of k-free clauses, but is easier to learn; formally, a single k-local clause is pac-learnable [Cohen, 1993b].

Due to space considerations, we will not give a lazy-macro specification of the bias.[15] Table 2 gives results with a k-local bias for a range of values of k, and compares these results with biases towards constrained, ij-determinate, and k-free clauses; the format is as for Table 1 except that standard errors for the error rates are also given. The k-local bias gives a statistically significantly lower error rate[16] than the k-free bias on the *loan* benchmark, and its error rates are statistically indistiguishable on the *family* bechmark.

[14] If a clause has no free variables, we will define its locality to be zero.

[15] In the specification, an antecedent is defined to be a sequence of locales, each containing different free variables, and the definition of a locale is similar to the definition of *kfree_lits*. The complete BRL consists of five macros and 24 lines of code.

[16] We obtained $t = 7.7$, $p > 0.995$ on an unpaired t-test between the 4-free and 2-local biases.

6 Rule schemata and other related work

Grendel's use of an explicit bias representation language is one of several techniques that have been proposed for constraining the search space of a learning system. CLINT [Raedt and Bruynooghe, 1992] and RDT [Kietz and Wrobel, 1992] use *rule schemata* (also called second-order schemata) to constrain search. An example of a rule schema is $Q(X,Y)$:- $P(X,Z) \wedge R(Z,Y)$ where Q, P, and R are *second order variables* that can be instantiated to any binary relation. Lazy macros can represent rule schemata quite easily: the schema above, for example, could be encoded[17] with the macro

```
body(Qxy) → [Pxz,Rzy] where
    predicate(Qxy), variables_of(Qxy,[X,Y]),
    predicate(Pxz), variables_of(Pxz,[X,Z]),
    predicate(Rzy), variables_of(Rzy,[Z,Y]).
```

To demonstrate this experimentally, we encoded the following four schemata using lazy macros.

S1. Q(X,Y) :- P(X,Y).
S2. Q(X,Y) :- P(Y,X).
S3. Q(X,Y) :- P(X,Z) ∧ R(Z,Y).
S4. Q(X,Y) :- P(Y,Z) ∧ R(Z,X).

This experiment was motivated by a result of Richards and Mooney [1992], who suggested the following bias: viewing the background predicates as edges in a graph, restrict the learner to output clauses that correspond to paths through this graph. Applying this "relational pathfinding" bias to the *family* benchmark, and additionally limiting paths to be of length k or less for $k = 2$, leads to the bias expressed by the four rule schemata above. The last lines of Table 2 confirm Richards and Mooney's results showing that this bias is appropriate for the *family* benchmark. Richards and Mooney also showed that a relational pathfinding bias is appropriate for qualitative modeling problems.

Another technique for reducing search was used in the MIS system [Shapiro, 1982]. The hypothesis space searched by MIS was defined by a "refinement operator": an arbitrary Prolog predicate that would, given a clause, return the possible "refinements" or specializations of that clause. Because the refinement operator is a separate module of the program, MIS's bias could be changed by replacing the refinement operator, much as Grendel's bias is changed by replacing its ADG. The refinement operator is clearly at least as general as an ADG. However, ADG's are a much more structured way of describing a bias; lazy macros are seemingly more powerful, as they allow an escape to Prolog, but are still highly constrained, as they must ultimately be expanded to ADG's. ADG's are also a more declarative description of the bias, as the refinement operator also encodes some control information.[18] For these reasons, ADG's, with or without lazy macros, are much easier to modify than refinement operators. We note that while only four refinement operators were implemented for MIS, Grendel has been used with over fifty different ADGs.

7 Concluding Remarks

To summarize, we have extended the bias description language of the Grendel learning system with a powerful macro facility; these "lazy macros" allow succinct specification of a number of biases that

[17] This certainly the most obvious encoding; however, it leads to a number of rules exponential in the length of the schema. More compact encodings (along the lines of the macros defining the k-free bias) are also possible.

[18] MIS used a top-down breadth-first search, guided by various oracles; the fundamental search step was to specialize a clause using a single application of the refinement operator. Grendel uses a top-down greedy search; the fundamental search step, however, is *not* to specialize a clause with a single grammar rule, but to specialize by applying sequences of grammar rules, guided by certain general heuristics.

are useful for inductive logic programming. To illustrate lazy macros, we presented specifications of several useful biases, including ij-determinate clauses, "k-free" clauses, and "k-local" clauses. In all cases the specifications were concise and have a declarative interpretation. Additionally, we have compared these biases experimentally on several benchmark problems. To summarize the results, the ij-determinacy bias does best on program-induction problems, while the k-local bias does best on the two problems requiring indeterminate variables. Additionally, a "relational pathfinding" bias implemented using rule schemata obtained good results on one of the two problems, but could not be applied to the other.

Some topics for further research remain. Since our bias description language also can describe the biases of theory-guided learning systems, the biases described in this paper should be easily integrated with theory-guided learning methods; however, this integration has not yet been carried out. Also, a problem with lazy macros is that they often produce ADG's that are impractically large; for example, the ADG for k-free clauses defines 2^k nonterminal symbols, one for every possible set of connected variables. This situation could be improved by allowing macros to be expanded as needed to rewrite a nonterminal symbol, rather expanding all macros before learning begins. One focus of our current research is developing a version of Grendel that allows this sort of "dynamic" expansion of macros.

Acknowledgments

The author wishes to thank Haym Hirsh for comments on a draft of this paper, and Mike Pazzani for providing the *loan* dataset.

References

(Cohen, 1991) William W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. AT&T Bell Labs Technical Memorandum, available from the author on request. To appear in *Artificial Intelligence*, 1991.

(Cohen, 1992) William W. Cohen. Compiling knowledge into an explicit bias. In *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen, Scotland, 1992. Morgan Kaufmann.

(Cohen, 1993a) William W. Cohen. Cryptographic limitations on learning one-clause logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Washington, D.C., 1993.

(Cohen, 1993b) William W. Cohen. Learnability of restricted logic programs. In *Proceedings of the Third International Workshop on Inductive Logic Programming*, Bled, Slovenia, 1993.

(Džeroski and Lavrac, 1991) Sašo Džeroski and N. Lavrac. Learning relations from noisy examples. In *Proceedings of the Eighth International Workshop on Machine Learning*, Ithaca, New York, 1991. Morgan Kaufmann.

(Džeroski *et al.*, 1992) Sašo Džeroski, Stephen Muggleton, and Stuart Russell. Pac-learnability of determinate logic programs. In *Proceedings of the 1992 Workshop on Computational Learning Theory*, Pittsburgh, Pennsylvania, 1992.

(Kietz and Wrobel, 1992) Jorg-Uwe Kietz and Stephan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In *Inductive Logic Programming.* Academic Press, 1992.

(Lavrač and Džeroski, 1992) Nada Lavrač and Sašo Džeroski. Background knowledge and declarative bias in inductive concept learning. In K. P. Jantke, editor, *Analogical and Inductive Inference: International Workshop AII'92.* Springer Verlag, Daghstuhl Castle, Germany, 1992. Lectures in Artificial Intelligence Series #642.

(Muggleton and Feng, 1992) Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Inductive Logic Programming.* Academic Press, 1992.

(Pazzani and Brunk, 1991) Michael Pazzani and Clifford Brunk. Detecting and correcting errors in rule-based exper systems: an integration of empirical and explanation-based learning. *Knowledge Acquisition*, 3:157–173, 1991.

(Pazzani and Kibler, 1992) Michael Pazzani and Dennis Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1), 1992.

(Quinlan, 1990) J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3), 1990.

(Quinlan, 1991) J. Ross Quinlan. Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*, Ithaca, New York, 1991. Morgan Kaufmann.

(Raedt and Bruynooghe, 1992) Luc De Raedt and Maurice Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8(2), 1992.

(Richards and Mooney, 1992) Bradley Richards and Ray Mooney. Learning relations by pathfinding. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, San Jose, California, 1992. MIT Press.

(Shapiro, 1982) Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.