File No. UIUCDCS-F-85-930

85-1

AQ-PROLOG:
A Prolog Implementation of an Attribute-Based Inductive Learning System

Jeffrey M. Becker

Department of Computer Science
University of Illinois
at Urbana-Champaign

January 1985

ISG 85-1

This work was supported in part by the National Science Foundation under grant NSF DCR 84-06801 and the Office of Naval Research under grant NOOO 14-82-K-0186.

Table of Contents

AQ-PROLOG:
A Prolog Implementation of an Attribute-Based Inductive Learning System

Abstract

This paper describes the Aq-Prolog program for learning discriminant descriptions of classes of objects from examples. It also describes the use of the program, and provides an evaluation of the program and the implementation task. This project is a test of the suitability of Prolog as an implementation language for programs based on the Aq.

1. Introduction

The first versions of the AQ-PROLOG program and this document were originally produced for a class project in CS-347, Knowledge Based Programming, under Prof. C. Sammut during the Fall 1983 Semester. The program is not itself knowledge based (except for variable type specifications) but represents a step in continuing research in machine learning oriented toward automated knowledge acquisition for intelligent systems. The program takes as input sets of example objects belonging to different classes and provides as output for each class a description which covers the examples of that class and none of the examples in any other class. This type of description is called a *discriminant description* since it may be used to determine to which class a given event belongs. Another version of Aq has been used to produce rules for use in an expert system with a high degree of success [Michalski et al, 1982].

Aq was implemented in UNSW Prolog [Sammut, 1983b] to determine if the language would be suitable for implementing programs based on the Aq algorithm and for doing related machine learning research. This evaluation is given in Section 4. The interest in Prolog is due in part to the selection of Prolog by the Japanese as (the basis of) the main language for the much publicized 5th generation computer project.

2. The Aq Algorithm

The Aq algorithm is based on a few simple concepts. This section gives a description of the upper levels of the Aq algorithm in a Pascal-like format and "plain English". For a more rigorous treatment of the technical details of the algorithm see [Michalski, 1975]. Aq-Prolog uses a subset of the VL1 representation language to represent concept descriptions [Michalski, 1974].

2.1. Terminology

First, terminology needed to understand the Aq methodology is defined:

Attribute

An *attribute*, also frequently called a *variable*, is some aspect of an object to be described such as color, temperature, shape, weight, etc. Associated with each attribute is a set of values which the attribute may assume called its *domain*. Three types of domains are recognized in Aq-Prolog: A *nominal* domain specifies a set of unordered discrete values which an attribute may have. For example color may be red, green, orange, etc. A *linear* domain specifies a set of ordered discrete values which an attribute may have. For example temperature may be 32 .. 105 (degrees). The ".." is the "range operator" and is used to specify a segment of values as above. A *structured* domain specifies a set of unordered discrete values in a tree structure, where the tree structure represents an abstraction hierarchy of descriptions. For example, "pentagon", "hexagon", and "octagon" may all be siblings of the node "polygon" for a "shape" attribute.

Selector

A *selector* consists of an attribute, a relational operator, and a single value or a disjunctive list of values. A selector in Aq-Prolog is a predicate which is either satisfied (true) or not satisfied (false) in a particular context. The selector [color = red] would be read "the color is red" for the particular object in question. The selector [shape = round, pointed] would be read "the shape is round or pointed". This is an example of "internal disjunction". Internal disjunction is not allowed in selectors for structured domain attributes, but is allowed for nominal and linear selectors. Other relations besides "=" are formally supported in VL1, but not in Aq-Prolog at this time.

Complex

A *complex* formally speaking is a cover of a portion of an event space. In Aq-Prolog a complex is a conjunction of selectors which cover a portion of the event space defined by all of the attributes declared for the current problem. The complex [color = red][shape = round, pointed] is a description for all objects which are red and have either a round or pointed shape.

Event

An *event* is a complex which describes a single point in the event space for the current problem. If the attributes color, shape, and weight are defined with appropriate domains, the complex [color = red][shape = round][weight = 25] would be an event. Events are also frequently referred to as *examples* since they serve as input examples of classes of objects for inductive learning algorithms.

Star A *star* is a set of all possible maximally general descriptions of a given event which don't cover some specified set of negative events. In the current context, a generalization of an event is always a complex. Since in practice we are only interested in generalizations of an event which satisfy certain criteria, such as covering a large number of the other events in the same class and brevity, a star is usually reduced in size by eliminating uninteresting generalizations.

Cover

A *cover* in Aq-Prolog is a disjunction of complexes.

2.2. Description of the Aq Algorithm

The Aq algorithm will be described in top-down fashion. The segments described here correspond to segments in the actual program code. The reader is encouraged to refer to the program listing in Appendices B and C while studying the algorithm.

The Aq algorithm is stated here in a Pascal-like format:

```
AQ(PosEvents : EventList, NegEvents : EventList) : Cover ;

 VAR
    UnCoveredEvents  : EventList ;
    SeedList         : EventList ;
    Seed             : Event ;
    Star             : CompSet ;
    BestComplex      : Complex ;
    Cover            : Cover ;

 BEGIN
  UnCoveredEvents := PosEvents ;
  SeedList := PosEvents ;
  Cover := NIL ;

  WHILE (UnCoveredEvents <> NIL) DO
   BEGIN
       Seed := FIRST(SeedList) ;
       Star := STAR(Seed, NegEvents, LEF) ;
       BestComplex := BEST(Star, LEF) ;
       UnCoveredEvents := KNOCKOUT1(BestComplex, UnCoveredEvents);
       SeedList := KNOCKOUT(Star, Seedlist) ;
       IF SeedList = NIL THEN SeedList := UnCoveredEvents ;
       Cover := DISJUNCTION-OF(Cover, BestComplex) ;
   END ; (* While *)

  RETURN(Cover) ;

END; (* Aq *)
```

The Aq algorithm picks one event as a seed and calls procedure STAR to generate a set of alternative generalizations of the seed event. The best one of these complexes is picked to become part of the cover. Any complex covered by the best one is removed from UnCoveredEvents, and any complex in SeedList covered by any complex in Star is removed from SeedList using the KNOCKOUT1 and KNOCKOUT procedures respectively. This is repeated until there are no UnCoveredEvents remaining. Provisions may also be made for counting the number of times the WHILE loop is executed after SeedList first becomes NIL to determine the number of complexes above the optimal number which are in the cover. The "Bound" parameter in the Prolog code allows for providing an initial hypothesis other than "NIL" to the Star algorithm - a feature useful for incremental learning but not currently used.

3.3. Description of the Star Algorithm

The Star generation procedure is stated here in a Pascal-like format:

```
STAR (EPlus : Event, NegEvents : EventList, LEF : Lef) : CompSet ;

 VAR Star : CompSet ;

 BEGIN
  Star := NIL ; (* "Initial hypothesis" - the entire event space *)

  FOR EMinus in NegEvents DO
   BEGIN
      XList := EXTENDAGAINST(EPlus, EMinus) ;
      Star := MULTIPLY(Star, XList, EMinus) ;
      Star := TRUNCATE(Star, LEF) ;
   END ;

  RETURN(Product) ;

 END ; (* Star *)
```

"Extension against" is the inductive generalization method used in the Aq algorithm. EXTENDAGAINST(EPlus, EMinus) returns a list of single selector complexes which are extensions of each selector in EPlus against the corresponding selector in EMinus. The extension of complex X against complex Y, denoted here as extendagainst(X, Y), is only defined if the intersection of X and Y is null. Extension against is defined differently for each of the different types of domains to reflect the differing semantics:

Nominal:

extendagainst([V = A], [V = B]) is [V = C]

where C is not(B), i.e., all points in the valueset of V which are not in B.

Linear:

extendagainst([V = A], [V = B]) is [V = L .. H]

where
1) if A < B then
   L = the low bound of the range of V, and
   H = B - 1.
2) if A > B then
   L = B + 1, and
   H = the high bound of the range of V.

Structured:

extendagainst([V = A], [V = B]) is [V = C]

where C is the highest ancestor of A which is not an ancestor of B.

"Multiply" returns a new Star in which no complex covers EMinus. This is accomplished by multiplying every complex in Star which has a non-null intersection with EMinus by every complex in XList,

producing a (possibly) much longer list of complexes. The test with EMinus is performed to eliminate unnecessary product computations, thus saving considerable computation time.

The Star algorithm performs a "beam search" through the space of alternative generalizations of EPlus. The complexes in the Star start out general (initially only one complex = entire event space) and become more specialized as extended complexes from EXTENDAGAINST(EPlus, EMinus) are multiplied in. The Star is truncated on each iteration to be no greater than a certain maximum size "Max-Star" in length by removing the least desirable complexes. By default, desirable complexes cover a large number of positive events and have few selectors. Other criteria may be specified by changing the evaluation functions and tolerances in the lexicographic evaluation function (LEF). It is also desirable to periodically perform an "absorption" operation on the Star to remove any complex which is covered by some other complex in the Star, thus eliminating redundancies.

The reader should now have a good understanding of how the program works. If more detailed understanding is desired the reader is welcome to study the code and/or run the program using the tracing facilities built into UNSW Prolog. It may also be enlightening to insert print statements into (a copy of) the code at key points such as after the "Star" subgoal in the "Aq" function. The next section describes how to prepare data sets for the program and how to run it.

3. User's Guide

Aq-Prolog is a relatively simple implementation of Aq (due to limits on time available for development) and thus has few options. This characteristic also makes the program quite easy to use. Some understanding of the syntax of Prolog would be helpful since all input is in the form of Prolog assertions or commands. Data sets may be prepared using any of the Unix system editors. Once a data set is prepared the program may be invoked by typing "prolog -sxxxx aqp". This assumes that the user has an alias for the UNSW Prolog system in his ".login" file and "aqp" is in the current working directory - see Appendix A for details. "-sxxxx" is used to instruct the Prolog interpreter to allocate additional stack space. Small data sets have been run successfully with a stack size of 3000. Stack sizes of 10000 or more may be needed. If the program aborts due to a stack overflow, restart Prolog using a larger stack size after ensuring that there are no errors in the data set.

3.1. Data Sets

A data set is a collection of Prolog assertions stored on a file. This file may be created using any of the standard Unix text editors. A data set consists of three sections: 1) type declarations, 2) a list of classes, and 3) training events for each class. All identifiers must begin with lower case letters so that they will not be interpreted by Prolog as unbound variables. Comments may be included in the file by delimiting them with "%"'s. There are no restrictions on the size of any domain, the number of classes, or the number of events per class. However, large problems may require a large amount of stack space and computer time. See the "Example" section for an example of a small data file.

3.1.1. Type Declarations

The information required for a type declaration varies according to the type of domain:

Meta-symbols used here are:

<var>	is a user-defined variable name.
val_i	is any atomic value (integer or constant).
<low>, <high>	are integers, <low> must be less than <high>.
[sibs]	is a list of sibling nodes in a structured domain.
<parent>	is a parent node in a structured domain.

Nominal:

domaintype(<var>, nominal).
valueset(<var>, [val_1, val_2, ... , val_n]).

Linear (integer):

domaintype(<var>, linear).
range(<var>, <low>, <high>).

Linear (symbolic):

domaintype(<var>, linear).
order(<var>, [val_1, val_2, ... , val_n]).

Structured:

domaintype(<var>, structured).
structure(<var>,
[parent([sibs], <parent>),
parent([sibs], <parent>),
.
.]).

note:

No sibling may have more than one parent in a structured domain. Variable names must be unique. "val" names may be re-used in different domains and will have a unique meaning in each.

3.1.2. Classes

A list of classes must be specified in the data set. The order of the class names in the list given will be the order in which class covers are generated by the Aq algorithm. Covers may tend to be lopsided, that is the description of the first class covered will be simpler and will cover a proportionately large area of the event space than subsequent descriptions, thus the order is significant. Classes are specified by the assertion:

classes([class-name$_1$, class-name$_2$, ...]).

3.1.3. Training Events

For each class name specified in the "classes" declaration there must be a set of training events from which the class description is to be learned. The set of training events for a class is specified by the assertion:

events(class-name, [<list-of-events>]).

where

list-of-events is	[$event_1$, $event_2$, $event_3$, ...],
each $event_i$ is	[sel_1, sel_2, sel_3, ...],
each sel_i is	[<var> = <val>].

for example,

events(car, [[[color = red], [doors = 2],[drive = front-wheel]],
[[color = blue],[doors = 3],[drive = four-wheel]],
[[color = blue],[doors = 4],[drive = rear-wheel]]]).

represents a set of training examples for the class "car".

Note: The commas are necessary. Be careful to balance brackets.

3.2. How To Run Aq-Prolog

Aq-Prolog allows the user to experiment with data sets easily. Data sets may be edited from within UNSW Prolog using the "ef" (NOT "em") command. The "ef <filename>!" command allows any file to be edited from within Prolog. A data set may be loaded or reloaded as described below.

3.2.1. Loading a data set

Once Aq-Prolog has been loaded a data set must be loaded into the Prolog system. This is done by typing "data(<filename>)!" where <filename> is the name of the file containing a data set created according to the specification given above. After running Aq on one data set, another may be loaded by typing "data(<filename>)!" again with the new file name. This will clear (*retract* in Prolog terminology) the old data set and load the new one. Initialization operations are performed during data set loading which may take a few moments on large data sets.

3.2.2. Invoking the Aq program

After loading a data set, the Aq program is invoked by typing "run!". Two "questions" will then be asked to set program parameters.

3.2.3. Setting MaxStar

The MaxStar parameter controls the number of alternative descriptions which are maintained during star generation. The user must experiment with the system to determine what values for MaxStar give good results for the data set being used. After the user types "run!" Aq-Prolog will prompt with:

Enter MaxStar for the current run >

Enter an integer number greater than zero. The upper bound on this parameter is determined by space available to the program, and CPU time available for processing. Too large a value may cause a stack overflow, or use very large amounts of computer time. Too small a value will cause excessive trimming of the stars generated, which lowers the probability that a truly optimal cover will be found. Typical values for this parameter run from about 10 for small problems (tens of examples) to over 50 for large problems (hundreds of examples).

3.2.4. Operating mode selection

There are three "modes" in which the AQ algorithm may be applied to the event sets. The second "question" is -

Select mode of operation by entering a number:
ic: Intersecting covers
dc: Disjoint covers
vl: VL mode (sequential)
>

Intersecting covers are generated by applying AQ in the following manner: let E_i represent the set of events to be covered for class i, and F be the set of training events specified for all other classes, i.e. $\cup\{E_j\}$, $j \neq i$. C_i is constructed by applying AQ to E_i against F. Thus, the intersection of any two covers $C_i \cap C_j$, $i \neq j$ may be non-null. The intersection will *not* contain any event points originally specified as training events, it only can occur over unspecified events.

Disjoint covers are generated by applying AQ in this manner: Let C_i = <cover of class i> and E_i = <training events for class i>. The cover for class i is generated by applying AQ to E_i against $\cup\{D_j\}$, $j \neq i$, where D_j is C_j if a cover has been generated for class j and E_j otherwise. In this manner, it is guaranteed that $C_i \cap C_j = \{\}$, $i \neq j$.

The "VL" or sequential mode produces covers in the following manner: The cover C_i for class i is produced by applying AQ to E_i against $\cup\{E_j\}$, $j > i$ in classname order. To utilize the covers produced by this mode, they must be tested in the same sequential order that they were constructed (i.e., classname order specified in the data set). Each cover may contain any event points allocated to a previously generated cover. This mode is useful in classifying new events with a minimum of variable testing, since fewer complexes are needed to specify some covers.

After the mode is selected, AQ will be applied as described, and the covers will be computed and printed. After all of the covers are printed the following question is printed:

Would you like to try another mode (yes,no) ? >

Enter "yes" if you want to try another mode on the same data, otherwise enter "no". This allows you to try any of the three modes on the same data. If "yes" is entered, the "Mode of operation?" prompt is re-issued. Otherwise, the program prints "Ok" and returns to the top level of Prolog.

3.3. Program Output

Aq-Prolog produces its covers as a series of disjunctive VL1 formulas for each class. Each is printed in VL1 format. The output appears as:

Cover of Class <first class name in specified order>:
[attr rel val][attr rel val] ... [attr rel val]
[attr rel val][attr rel val] ... [attr rel val]
.
.
.

where each line represents a separate interval (complex). The cover is the union of all intervals printed under the heading line. This is repeated for each class covered by AQ. If no intervals are printed, the cover is the entire event space.

3.4. Example

The following is a typescript file of an actual session using Aq-Prolog. Added comments are in "{}". For more information on the site-dependent details of running UNSW Prolog and Aq-Prolog, see Appendix A.

```
% more data2                    {look at the data file}
% trial data set %              {<- this is a comment in the data file}


domaintype(color, nominal).
valueset(color, [red, green, blue]).

domaintype(temp, linear).
order(temp, [cold, warm, hot]).
```

```
domaintype(shape, nominal).
valueset(shape, [square, hexagon, octagon]).


classes([past, present, future]).

events(past, [ [[color = red],[temp = warm],[shape = square]],
          [[color = green],[temp = cold],[shape = hexagon]] ]).

events(present, [ [[color = blue],[temp = warm],[shape = square]],
            [[color = red],[temp = hot], [shape = hexagon]] ]).

events(future, [ [[color = green],[temp = warm],[shape = hexagon]],
            [[color = blue],[temp = hot], [shape = octagon]] ]).


% prolog -s5000 aqp              {invoke UNSW Prolog}


Aq-Prolog [Version 1.0]          {Aq-Prolog signon messages}

Type 'data(<filename>)!' to load a data file
Type 'run!' to start Aq


UNSW - PROLOG
: data(data2)!                   {load a data set}
Loading data2.
: run!                           {invoke aq}

Enter MaxStar for the current run > 10


Select mode of operation by entering ic, dc, or vl:
  ic: Intersecting Covers
  dc: Disjoint Covers
  vl: VL mode (sequential)

> ic

CPU Time = 20.30 secs


Cover of class past:

[color = red][shape = square]
[temp = cold]


Cover of class present:

[color = blue][shape = square]
```

[color = red][shape = hexagon]

Cover of class future:

[color = blue v green][shape = hexagon v octagon][temp = warm .. hot]

Would you like to try another mode (yes,no) ? > yes

Select mode of operation by entering ic, dc, or vl:
  ic: Intersecting Covers
  dc: Disjoint Covers
  vl: VL mode (sequential)

> dc

CPU Time = 21.63 secs

Cover of class past:

[color = red][shape = square]
[temp = cold]

Cover of class present:

[color = blue][temp = warm]
[shape = hexagon][temp = hot]

Cover of class future:

[color = green][temp = warm]
[shape = octagon][temp = hot]

Would you like to try another mode (yes,no) ? > yes

Select mode of operation by entering ic, dc, or vl:
  ic: Intersecting Covers
  dc: Disjoint Covers
  vl: VL mode (sequential)

> vl

CPU Time = 8.13 secs

Cover of class past:

[color = red][shape = square]
[temp = cold]

Cover of class present:

[color = blue v red][shape = hexagon v square]

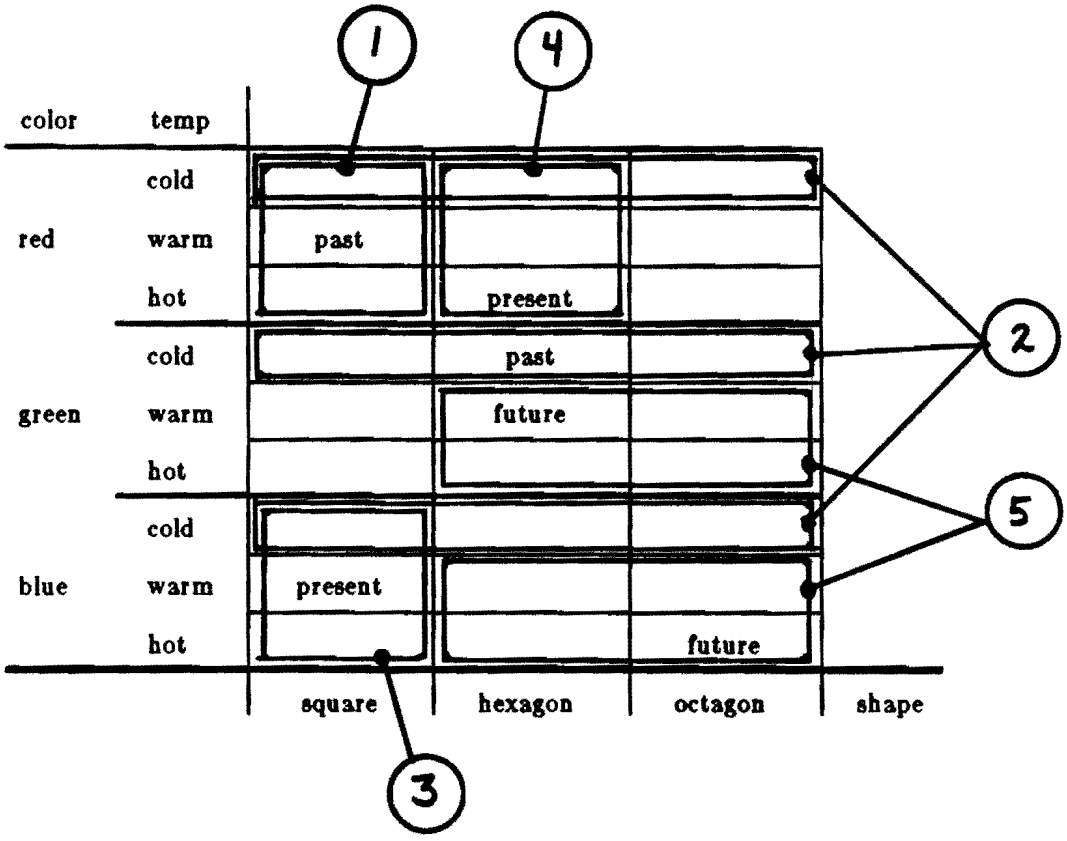
Cover of class future:

Would you like to try another mode (yes,no) ? > no
Ok
: ^D

3.5. Generalized Logic Diagrams

Generalized Logic Diagrams (GLD's) are used to graphically display covers in an event space for a multiple-valued logic such as VL1 [Michalski, 1971]. Each square in a GLD represents one of the possible events in the event space defined by the domains of the variables in the given data set. In the diagram for Intersecting Covers, note that covers for different classes intersect only in don't-care space, as should be expected. In the diagram for Disjoint Covers, note that complexes belonging to the cover of a class may overlap. In VL mode, a class cover may cover events in a preceding class, and the last class cover generated will always cover the entire event space. GLD's are given here for the covers produced in the above example. The numbers left of the VL1 descriptions correspond to the circled numbers in the diagrams.

Intersecting Covers



Cover of class past:

1. [color = red][shape = square]
2. [temp = cold]

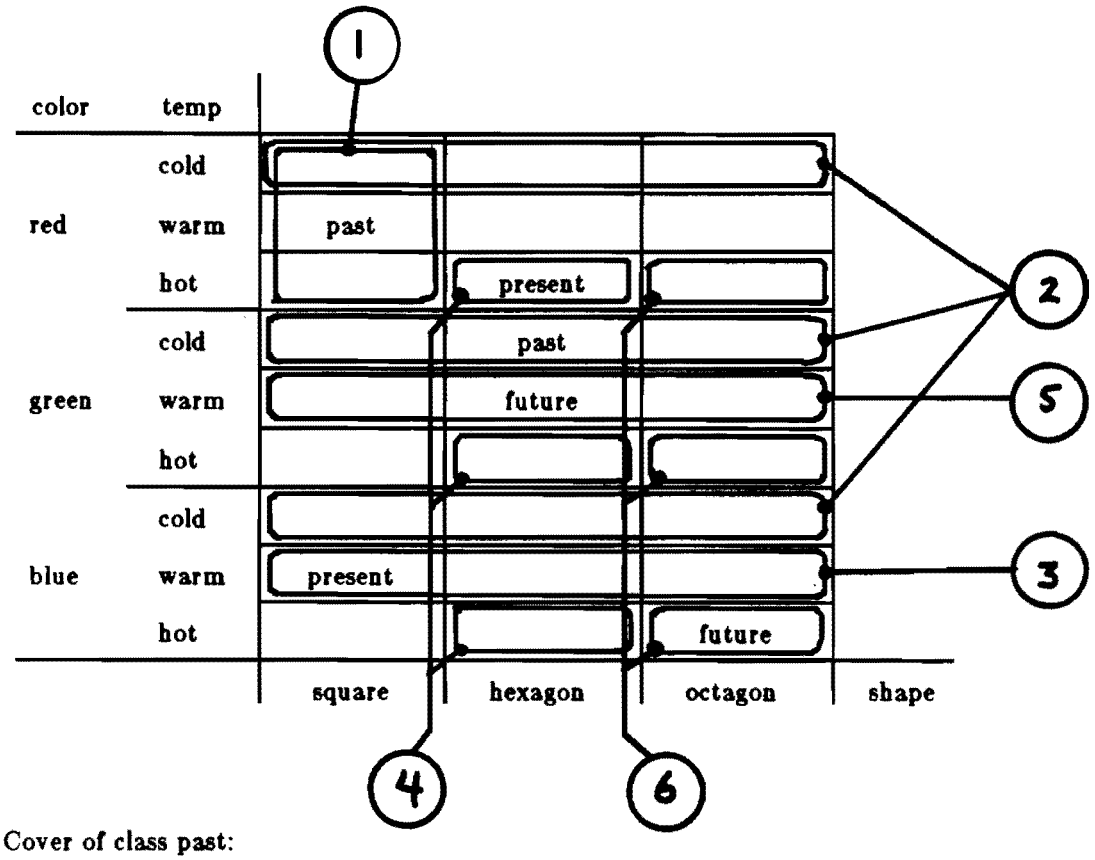
Cover of class present:

3. [color = blue][shape = square]
4. [color = red][shape = hexagon]

Cover of class future:

5. [color = blue v green][shape = hexagon v octagon][temp = warm .. hot]

Disjoint Covers



Cover of class past:

1. [color = red][shape = square]
2. [temp = cold]

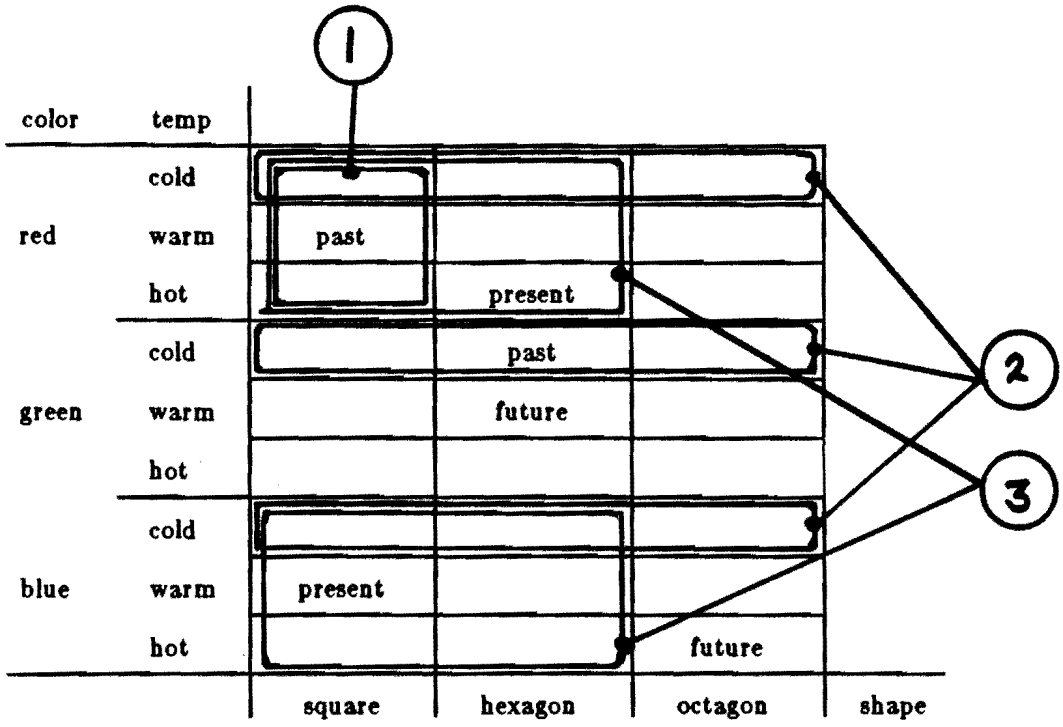
Cover of class present:

3. [color = blue][temp = warm]
4. [shape = hexagon][temp = hot]

Cover of class future:

5. [color = green][temp = warm]
6. [shape = octagon][temp = hot]

VL mode (sequential) covers



Cover of class past:

1. [color = red][shape = square]
2. [temp = cold]

Cover of class present:

3. [color = blue v red][shape = hexagon v square]

Cover of class future:

(entire event space)

4. Evaluation

Aq-Prolog is based primarily on AQINTERLISP, a similar implementation of Aq written in Interlisp [Becker, 1983]. AQINTERLISP has a number of additional features, the most notable being an interactive data set editor. Aq-Prolog and AQINTERLISP were run on identical data sets to compare execution times. The data set "data2", which is used in the preceding example was used. MaxStar was set at 10. A stack size of 5000 was not adequate so a stack size of 10,000 was used for running Aq-Prolog on this (small) data set. The UNSW "cputime" primitive was used for timing Prolog and the built in Interlisp timing facility was used for timing Interlisp. Both programs were run on Vax-B at the University of Illinois Department of Computer Science. Times are given for both the interpreted and compiled versions of AQINTERLISP. No compiler is available for UNSW Prolog at this time.

TIME (CPU Seconds)

Mode	AQINTERLISP		Aq-Prolog
	interpreted	compiled	interpreted
1 (IC)	21	0.75	22
2 (DC)	11	0.40	24
3 (VL)	11	0.23	9

The performance of Aq-Prolog in terms of computation time is better than expected, especially since the program does not use a bit string representation for the right hand side of selectors. Trial runs on other data sets yielded similar results: Aq in UNSW Prolog runs as fast as in interpreted Interlisp. Another version of Aq in Prolog written by Thierry Schang which uses a bit string representation, called Gem-Prolog, runs around 2 to 3 times faster than Aq-Prolog, hence faster than interpreted AQINTERLISP. It should be pointed out that AQINTERLISP makes extensive use of CLISP macros which slows down interpretation compared to the same program written in "pure" lisp, but which usually speeds up compiled performance. As is documented above, compilation speeds up Interlisp programs considerably (by a factor of about 20 - 60 depending on how the program is coded). It will be interesting to see how compilation improves the performance of Aq-Prolog when a compiler becomes available for UNSW Prolog.

Prolog is a powerful language for applications requiring frequent pattern matching and a backtracking control facility. I found UNSW Prolog very easy to learn. I attribute this to the small number of reserved words in the language, the simple control structure, and the simple syntax. The major conceptual difficulties are understanding the backtracking scheme and the use of "cuts" for controlling backtracking. Prior knowledge of Lisp helps in understanding recursion, which is used extensively in Prolog, and Prolog list operations. It takes as little as two hours of study to begin using Prolog and about 6-10 hours of practice and exposure to code written by experts to develop a moderate degree of proficiency in the language.

Development of Aq-Prolog went rapidly. This is largely due to previous experience with the AQINTERLISP program. The (very rough) breakdown of development time is:

Design	15 Hrs.
Coding	19 Hrs.
Debugging &Testing	23 Hrs.
Total	57 Hrs.

No figures of similar development times for AQINTERLISP are available, nor would they be meaningful since the development cycle for that program was quite different. Design time is low since I have written

nearly the same program before in Interlisp. The internal representations of selectors are different, however, and support code for the basic representation makes up a significant proportion of the code (roughly 1/3). Coding in Prolog is "quick and easy", except as noted below, and the structure of Prolog encourages the programmer to think out and write algorithms in a clear and concise manner. Debugging time is not unusually high. Typically 1/3 to 1/2 of program development time is debugging and testing. The types of problems typically encountered were minor typos which caused subgoals to fail to be matched, and oversights in the initial coding. Minor typos are especially difficult to find, but the tracing facility provided in UNSW Prolog proved invaluable in tracking these down. The "em" (edit module) facility also proved to be quite a time saver.

Aq-Prolog consists of about 600 lines of code (excluding comments). AQINTERLISP consists of about 1200 lines of code (also excluding comments). The data set editing functions, as well as the top level menu and help functions were NOT counted in AQINTERLISP since there are no corresponding functions in Aq-Prolog. AQINTERLISP includes slightly fancier cover printing functions, a file descriptor, and initialization of global variables. Aq-Prolog includes low level list operations which are predefined in Interlisp. Qualitatively, the Prolog version seems to be somewhat easier to read and understand.

Prolog has some drawbacks. First is the slow speed, due in large part to the fact that it is an interpreted language. Interpreted AQINTERLISP is not useable for serious work, and neither is Aq-Prolog because of the slow speed. The Aq algorithm is a collection of iterative procedures. The "bagof" and "repeat" predicates are useful in many cases, but not all. The exceptions are written in Prolog using tail recursion and result in the need for large stack allocations. Both the tail recursion problem and the speed problem can be improved upon by a good compiler for Prolog. One problem that cannot be corrected by a compiler is the need to pass large numbers of parameters (which are local variables in iterative code) from one instantiation of a recursive function to the next (see "Aq" and "Star" in the program listing). Some form of iterative subgoals with local variables allocated on the stack for side effect operations (assignment) during iteration would make this type of procedure much more readable, writeable, and probably more efficient.

Overall, Aq-Prolog makes almost no use of the more powerful features of Prolog in any meaningful way. Backtracking is only used in several instantiations of "bagof" to map through the members of list and build another list, in "extendref" for structured domain attributes, and in matching linear domain symbols to ordinal values. Unification is used mainly as a simple parameter passing and list dissection mechanism. Most Prolog "predicates" are used as functions, with several of the parameters being input arguments, and one or two others being output values. This type of parameter passing can sometimes make code opaque.

It is not convenient to require that the user anticipate and specify the stack allocation prior to running Prolog. When running Aq-Prolog it is necessary to specify larger stack sizes when larger data sets are used, but there is no way of determining a priori how large the stack allocation should be. Much wasted CPU time results when a program working on a large problem nearing completion runs out of stack space! It is wasteful to always allocate huge stack spaces to avoid this problem. An automatic stack allocation mechanism is needed to make UNSW Prolog more friendly to large system builders.

5. Conclusions

The algorithm used in Aq-Prolog has been described. Operating instructions for the program and a sample run have been given. Analyses of speed, code size, programming effort and various qualitative aspects of Aq-Prolog have been given. Prolog is a powerful and simple language with many desirable features. "Mudball" languages such as Interlisp provide many more tools for making the programmers job easier (eg. Masterscope, iterative control constructs, piece-meal compilation, completely automatic storage management). No doubt similar features could be added to Prolog with similar development effort.

Suggested improvements for Aq-Prolog include improving the format of data sets to correspond more closely to the conventional notation for complexes, and adding statistics gathering features which would show the number of events covered by each complex in a cover. Also, it may be desirable to assign a cost to each variable, corresponding to the salience of the variable or real world measurement costs, and to use this cost as the basis of a quality criterion in the LEF. More advanced improvements might include speed and stack space optimization, background rules, and incremental cover generation capabilities. Detailed discussion of these topics is beyond the scope of this paper.

Acknowledgements

This work was supported in part by the National Science Foundation under grant NSF DCR 84-06801 and the Office of Naval Research under grant ONR NOOO 14-82-K-0186. Thanks go to Prof. C. Sammut for suggestions, Prolog system support, and constructive criticism; to Jeff Gilbert for help with revisions to the Prolog code; and to Prof. R. S. Michalski for many helpful comments on the contents of this document.

References

(1) – , Interlisp Reference Manual, Warren Teitelman, ed., Xerox Palo Alto Research Center, California, 1978.

(2) –,Interlisp-D Users Guide, Xerox Electro-Optical Systems, Pasadena, California, February 1982.

(3) Becker, J. M., "AQINTERLISP: An Interlisp Program for Inductive Generalization of VL1 Event Sets", File No. UIUCDCS-F-83-911, ISG 83-31, Department of Computer Science, University of Illinois, Urbana, September, 1983.

(4) Larson, James, and R. S. Michalski, "AQVAL/1 (AQ7) Users Guide and Program Description", Report number 731, Department of Computer Science, University of Illinois, Urbana, June 1975.

(5) Michalski, R. S., "A Geometric Model for the Synthesis of Interval Covers", Report number 461, Department of Computer Science, University of Illinois, Urbana, June, 1971.

(6) Michalski, R. S., "Variable-Valued Logic: System VL1", 1974 International Symposium on Multiple-Valued Logic, West Virginia University, Morgantown, West Virginia, May 1974.

(7) Michalski, R. S., "Synthesis of Optimal and Quasi-Optimal Variable-Valued Logic Formulas", ISG 1975-1, Proceedings of the 1975 International Symposium on Multiple-Valued Logic, Bloomington, Indiana, May 13-16, 1975, pp. 76-78.

(8) Michalski, R. S., and J. H. Davis, V. S. Bisht, J. B. Sinclair, "PLANT/ds: An Expert Consulting System for the Diagnosis of Soybean Diseases", Proceedings of the 1982 European Conference on Artificial Intelligence, Orsay, France, 12-14 July, 1982.

(9) Michalski, R. S., and Larson, J. B., "Selection of Most Representative Training Examples and Incremental Generation of VL1 Hypotheses: the underlying methodology and description of programs ESEL and AQ11", Report Number 867, Department of Computer Science, University of Illinois, Urbana, May 1978.

(10) Michalski, R. S., and McCormick, B. H., "Interval Generalization of Switching Theory", Report number 442, Department of Computer Science, University of Illinois, Urbana, May 1971.

(11) Sammut, R. A., and C. A. Sammut, "PROLOG: A Tutorial Introduction", The Australian Computer Journal, Vol. 15, No. 2, May 1983.

(12) Sammut, C. A., and R. A. Sammut, "The Implementation of UNSW-PROLOG", The Australian Computer Journal, Vol. 15, No. 2, May 1983.

Appendix A : Site-dependent details

The Unix environment variable $ISG should be set to

/mntb/2/michalski/isg

Aq-Prolog is currently stored on Vax-B at

$ISG/AQPROLOG/aqp

This document is stored on Vax-B at

$ISG/AQPROLOG/aqpdoc

Some sample data sets are

$ISG/AQPROLOG/datax
" /data1
" /data2

UNSW Prolog may be accessed easily by placing

alias prolog /mntb/2/michalski/isg/PROLOG/bin/prolog

in your ".login" file.

Appendix B : Program Guide

This appendix serves as a guide to the program listing in Appendix C. The reader should have a good working knowledge of Prolog before attempting serious study of the code. An overview of the organization of the program, an outine of the program which lists the major sections and program functions, and an alphabetically sorted index of all the program functions with page number locations, are given here.

The order of functions in the program listing is "top down." The higher level functions are given first, followed by the lower level functions. It is not always necessary to understand the details of the lower level functions in order to understand the higher level functions, as long as the purpose of the lower level functions is understood. A brief comment about the purpose of each function is given in the program listing. More detailed comments are given in the listing for major functions and code segments.

There are three major divisions in the code: Encoding, Running Aq, and Decoding. The encoding functions transform structures in a data set to internal format. The "running Aq" functions produce class covers from the encoded data set. The decoding functions pretty-print the results. Functions are written at four different levels: Comset, Complex, Selector, and Selector-Right-Hand-Side. Comset level functions deal with lists of complexes. Complex level functions deal with individual complexes, which are formed from a conjunction of selectors. Selector level functions deal with individual selectors. Selector-Right-Hand-Side functions deal with the disjunction of values on the right-hand-side of a selector. These classes of functions overlap somewhat. Closely related functions are kept adjacent in the code. There are also some miscellaneous utility functions.

Aq-Prolog Outline

Aq-Prolog Functions

Aq-Prolog Functions

Aq-Prolog Functions

Appendix C : Program Listing

```
%                       AQ-PROLOG
%
%                The Aq algorithm, written in Prolog.
%            See documentation in ~jbecker/aqprolog/aqpdoc
%                 for operating instructions.


% special operators used in Aq-Prolog

op(500, yfx, ..) !     % Range operator for linear selectors



%  AQ-PROLOG data set input.
%
%    Only one data set is loaded at a time.  The previous data set
%    is erased when a new one is loaded.  To load a data set type
%
%                          data(FileName)!
%
%    This will clear the old data set, record the file name for the
%    new data set, then read and process each clause in the data set.


data(FileName) :-
        clear,
        assert(dataset(FileName)),
        print("Loading ", FileName, "."),
        see(FileName),
        repeat,
        read(X),
        (eof -> seen, ! ;
                process(X), fail).



% process(data_set_clause).
%
% A data file is stored as a sequence of Prolog clauses.
% Each type of clause is processed and the appropriate information is
% asserted into the Prolog data base.  Structures are transformed from
% the somewhat more readable form used in the data set to an internal
% format which is more suitable for efficient computation.

process(domaintype(Attr, D)) :-
        assert(domaintype(Attr, D)), !.
process(valueset(Attr, Valset)) :-
```

```
        sort(Valset, Vals),
        assert(valueset(Attr, Vals)), !.
process(range(Attr, Low, High)) :-
        assert(range(Attr, Low, High)),
        assert(subtype(Attr, integer)), !.
process(order(Attr, Ord)) :-
        length(Ord, High),
        assert(order(Attr, Ord)),
        assert(range(Attr, 1, High)),
        assert(subtype(Attr, symbolic)), !.
process(structure(Attr, Struc)) :-
        explodestruc(Attr, Struc),
        assert(structure(Attr, Struc)), !.
process(classes(CList)) :-
        storeclasses(CList), !.
process(events(Class, EventList)) :-
        encodeevents(EventList, EncodedEvents),
        storeevents(Class, EncodedEvents), !.

% Invalid data set entry, print error message -

process(X) :-
        print("Invalid data set entry: ", X), !.




% storeclasses(List_of_Class_Names).
%
% Store the list of classnames as "class(ClassName)" assertions.

storeclasses(CList) :-
        member(ClassName, CList),
        assert(class(ClassName)),
        fail.
storeclasses(_).




% storevents(Class, Events_for_this_Class).
%
% Store the list of events associated with a class as
% "event(Class, Event)" assertions.

storeevents(Class, EventList) :-
        member(Event, EventList),
        assert(event(Class, Event)),
        fail.
storeevents(_, _).
```

```
% Clear all definitions associated with previous data set (if any).

clear :-
        dataset(X),
        retractall(domaintype(_ _)),
        retractall(valueset(_ _)),
        retractall(range(_ _ _)),
        retractall(order(_ _)),
        retractall(structure(_ _)),
        retractall(class(_)),
        retractall(event(_ _)),
        retractall(subtype(_ _)),
        retractall(ancestry(_ _ _)),
        print("Data set ", X, " cleared."),
        retractall(dataset(_)).
clear.



% listdata
%
% List the currently loaded data set in pretty-printed format.

listdata :-
        nl,
        dataset(DataSetName),
        print("Data set ", DataSetName, ":"),
        nl,
        printdomaininfo,
        nl,
        printevents, !.



% printdomaininfo
%
% Print the variables in the current data set and their domaintypes.

printdomaininfo :-
        domaintype(Var, DType),
        print("Variable ", Var, "          has type ", DType, "."),
        fail.
printdomaininfo.
```

```
% printevents
%
% Print the set of events for the currently loaded data set.

printevents :-
        class(Class),
        event(Class, Event),
        princomplex(Event),
        print(" ::> ", Class),
        fail.
printevents.
```

```
% TOP LEVEL DRIVER FOR THE AQ ALGORITHM.
% Prompts user for maximum star size and mode of operation, then invokes
% the Aq algorithm on the currently loaded data set.

% top level entry point

run :-
        nl,
        ask("Enter MaxStar for the current run > ", MaxStar),
        nl,
        repeat,
        nl,
        print("Select mode of operation by entering ic, dc, or vl:"),
        print("  ic: Intersecting Covers"),
        print("  dc: Disjoint Covers"),
        print("  vl: VL mode (sequential)"),
        nl,
        ratom(Mode),
        retractall(cover(_ _)),
        nl,
        time(makecovers(Mode, MaxStar)),
        showcovers,
        nl, nl, nl,
        ask("Would you like to try another mode (yes,no) ? > ", Retry),
        ((Retry = yes) -> fail ; !),
        print("Ok"), !.
```

```
% makecovers(Mode, MaxStar).
%
% Makes the cover for each class in the indicated mode.
% The positive and negative events (or complexes) for a give class are
% collected, then the Aq algorithm is invoked.
% The cover is then stored for future reference.


% Make covers in the Intersecting Covers mode

makecovers(ic, MaxStar) :-
        class(Class),
        posevents(Class, EPos),
        negevents(Class, ENeg),
        aq(EPos, ENeg, EPos, EPos, MaxStar, [[]], Cover),
        storecover(Class, Cover),
        fail.
makecovers(ic, _).



% Make covers in the Disjoint Covers mode

makecovers(dc, MaxStar) :-
        class(Class),
        posevents(Class, EPos),
        neg_cover_or_events(Class, ENeg),
        aq(EPos, ENeg, EPos, EPos, MaxStar, [[]], Cover),
        storecover(Class, Cover),
        fail.
makecovers(dc, _).



% Make covers in the VL (sequential mode)

makecovers(vl, MaxStar) :-
        class(Class),
        posevents(Class, EPos),
        followingevents(Class, ENeg),
        aq(EPos, ENeg, EPos, EPos, MaxStar, [[]], Cover),
        storecover(Class, Cover),
        fail.
makecovers(vl, _).
```

```
% Error Message for error in mode selection

makecovers(X, _, _, _) :-
        nl,
        print("Enter ic, dc, or vl when making the mode selection"),
        abort.



% Store the cover of a class as "cover(Class, Complex)" assertions

storecover(Class, Cover) :-
        member(Complex, Cover),
        assert(cover(Class, Complex)),
        fail.
storecover(_, _).



% Collect the positive events for Class, return in "EPos"

posevents(Class, EPos) :-
        bagof(Event, event(Class, Event), EPos), !.



% Collect the negative events for a Class, return in "ENeg"

negevents(Class, ENeg) :-
        bagof(Event, negevent(Class, Event), ENeg), !.



% "Event" is a negative event for "Class"

negevent(Class, Event) :-
        event(NegClass, Event),
        not(NegClass = Class).



% Pick out a covering complex of class, or if none, an event of class

cover_or_event(Class, Comp) :-
        cover(Class, Comp).
cover_or_event(Class, Comp) :-
        event(Class, Comp).
```

```
% Pick out covers or events for a given class

pos_cover_or_events(Class, PosComps) :-
        bagof(Comp, cover_or_event(Class, Comp), PosComps).



% Pick out covers or events of all negative classes

neg_cover_or_events(Class, NegComps) :-
        bagof(Comp, negcomp(Class, Comp), NegComps).


% Pick out a covering complex from a negative class

negcomp(Class, Comp) :-
        cover_or_event(NegClass, Comp),
        not(NegClass = Class).



% Pick out events of all classes following given class

followingevents(Class, SEvents) :-
        bagof(ClassName, class(ClassName), Classes),
        following(Class, Classes, FClasses),
        bagof(Event, followev(FClasses, Event), SEvents), !.

followev(FClasses, Event) :-
        member(Class, FClasses),
        event(Class, Event).
```

```
%
%                    COMSET LEVEL OPERATIONS
%



%                              AQ
%
% aq(EList, FList, UnCovered, SeedList, MaxStar, Bound, Result)
%
% EList       - List of events to be covered (positive examples)
% FList       - List of events NOT to be covered (negative examples)
% UnCovered   - Positive events not yet covered
% SeedList    - List of events used as seed events for star generation
% MaxStar     - Maximum star size
% Bound       - The initial hypothesis (usually the entire event space: [])
% Result      - The Cover of EList against FList

aq(_, _, [], _, _, _, []) :- !.
aq(EList, FList, UnCovered, [], MaxStar, Bound, Result) :- !,
        aq(EList, FList, UnCovered, UnCovered, MaxStar, Bound, Result).
aq(EList, FList, UnCovered, SeedList, MaxStar, Bound, [Best | Cover]) :- !,
        first(SeedList, Seed),
        star(Seed, FList, MaxStar, [EList, UnCovered], Bound, Star),
        lef(LEF),
        selectbest(Star, 1, LEF, [EList, UnCovered], [BestComp]),
        coveredbycomplex(BestComp, UnCovered, CoveredEvents),
        trim(BestComp, CoveredEvents, Best),
        knockout1(Best, UnCovered, NewUnCovered),
        knockout(Star, SeedList, NewSeedList),
        aq(EList, FList, NewUnCovered, NewSeedList, MaxStar, Bound, Cover).
```

```
%                    STAR
%
% Create a Star of alternative generalizations of "Seed" which cover
% "Seed" and do not cover any of the negative events in "FList".
% The number of alternative generalizations maintained is limited by
% MaxStar.
%
% star(Seed, FList, MaxStar, LEFargs, Bound, Result)
%
% Seed        - The seed event which is generalized to create a star of
%               alternative generalizations.
% FList       - A list of negative events which must not be covered.
% MaxStar     - Maximum star size.
% LEFargs     - Arguments used by the evaluation functions in the LEF
% Bound       - Initial hypothesis for star building (usually "[]")
% Result      - A trimmed star of alternative generalizations.

star( _, [], _, _, PStar, PStar) :- !.
star(E, [F | FTail], MaxStar, LEFargs, PStar, NewPStar) :- !,
        extendagainst(E, F, EStar),
        multiply(PStar, EStar, F, EPStar),
        absorb(EPStar, MaxStar, APStar),
        lef(LEF),
        selectbest(APStar, MaxStar, LEF, LEFargs, ReducedStar),
        star(E, FTail, MaxStar, LEFargs, ReducedStar, NewPStar).
```

```
% multiply(ComSet, PStar, NegEvent, EPStar)
%
% Multiply a ComSet, which is a bounded star, by a PStar, which
% is a list of single-selector complexes formed by extension against
% NegEvent. Returns an expanded ComSet which is the result of multiply-
% ing each complex in ComSet which covers NegEvent by each complex in
% PStar.
%
% ComSet      - a star of complexes
% PStar       - a star of single-selector complexes
% NegEvent    - the negative event used in forming PStar
% EPStar      - the result - a star of complexes, none of which covers
%               NegEvent.

multiply(ComSet, PStar, NegEvent, EPStar) :-
        bagof(NewComps,
                (member(Comp,ComSet),
               (disjointcomps(Comp, NegEvent) ->
                 NewComps = [Comp], ! ;
                   bagof(A, (member(P,PStar), product(Comp,P,A)), NewComps),! )),
          EPList),
        appendx(EPList, EPStar), !.



% absorb(Star, MaxStar, AStar)
%
% Remove complexes from a Star which are redundant with respect
% to inclusion, ie. remove all complexes which are covered by some
% other complex in the Star. Done only if length of Star > MaxStar.
%
% Star        - a star of complexes
% MaxStar     - the maximum number of complexes allowed in a star
% AStar       - the result - an absorbed star

absorb(Star, MaxStar, AStar) :-
        length(Star, N),
        N > MaxStar, !,
        absourbr(Star, [], Star1),
        absourbr(Star1, [], AStar).
absorb(Star, _, Star).
```

```
% absourbr(Star, Buff, ARStar)
%
% Performs uni-directional absorption and reverses the
% remaining complexes in Star. Reversed absorbed Star
% is accumulated in "Buff" and returned in "ARStar".

absourbr([], S, S) :- !.
absourbr([C | S], B, ARStar) :- !,
        knockout1(C, S, Rs),
        absourbr(Rs, [C | B], ARStar).
```

```
% selectbest(PStar, MaxSize, LEF, LEFargs, ReducedStar)
%
% Select the "best" complexes from a list of complexes
% according to the criteria and tolerances given in a LEF.
%
% PStar         - a partial star of complexes
% MaxSize       - the maximum size that PStar should be
% LEF           - lexicographic evaluation function
% LEFargs       - arguments for evaluation functions used in LEF
% ReducedStar           - the result - best complexes from PStar


selectbest(PStar, MaxSize, _, _, PStar) :-
        length(PStar, L),
        L <= MaxSize, !.
selectbest(PStar, MaxSize, [CT | CTx], LEFargs, ReducedStar) :- !,
        reduce(PStar, CT, LEFargs, RStar),
        selectbest(RStar, MaxSize, CTx, LEFargs, ReducedStar).
selectbest(PStar, MaxSize, [], _, RStar) :-
        firstn(PStar, MaxSize, RStar), !.
```

```
% reduce(PStar, C&T, LEFArgs, ReducedStar)
%
% Reduce a star of complexes in size by applying a criterion from the
% LEF and keeping only those complexes which satisfy the tolerance.
%
% PStar                 - the star of complexes to be reduced
% C&T           - Criterion function and tolerance. Tolerance is
%                 expressed as a numerator "N" and a denominator "D"
% LEFArgs       - arguments for criterion function
% ReducedStar           - the result - best remaining complexes form PStar

reduce(PStar, [CritFn, N, D], [EPlus, UnCovEPlus], RStar) :-
        bagof(VC,
                (member(Comp, PStar),
                 CritFn(Comp, EPlus, UnCovEPlus, V),
                 VC = [V, Comp]),
            VList),
        minmax(VList, Min, Max),
        Tol is Min + (N * (Max - Min))/D,
        bagof(C,
                (member([V, C], VList),
                 V <= Tol),
            RStar), !.


% minmax(VCList, Min, Max)
%
% Find the Min and Max value in VCList which is a
% list of  [Value, Complex] pairs.

minmax([[X,_], [Y,_] | R], Min, Max) :-
        X <= Y, !,
        lohi([X, Y | R], Min, Max).
minmax([[X,_], [Y,_] | R], Min, Max) :- !,
        lohi([Y, X | R], Min, Max).

%
lohi([X, Y, [Z,_] | R], Min, Max) :- !,
        min([X, Z], A),
        max([Y, Z], B),
        lohi([A, B | R], Min, Max).
lohi([X, Y], X, Y) :- !.
```

```
%
%           Criterion Functions used in LEF
%

% Count the number of events covered by a complex. The NEGATIVE of
% this value is used since it is desirable have many events covered.

numbercovered(Comp, _, Events, N) :-
        coveredbycomplex(Comp, Events, CoveredE),
        length(CoveredE, P),
        N is -P, !.


% Count the number of selectors in a complex

numberofselectors(Comp, _, _, N) :-
        length(Comp, N), !.


% LEF, consists of (CriterionFunction, Numerator, Denominator) triples
%     Numerator <= Denominator, ie. a fraction between 0 and 1.

lef([[numbercovered, 0, 1], [numberofselectors, 0 , 1]]).
```

```
% knockout(OuterComps, InnerComps, UnCovComps)
%
%  Returns a list of all complexes in InnerComps NOT covered by
%  some complex in OuterComps. UnCovComps is the result.

knockout(OuterComps, InnerComps, UnCovComps) :- !,
        bagof(InComp,
                (member(InComp, InnerComps),
                 nonecover(OuterComps, InComp)),
              UnCovComps).



% nonecover(OuterComps, InComp)
%
%   Succedes if no complex in the list of complexes OuterComps covers
%   the complex InComp,  fails otherwise.

nonecover([], InComp) :- !.
nonecover([Comp | Cx], InComp) :- !,
        not(covers(Comp, InComp)),
        nonecover(Cx, InComp).



% knockout1 - like knockout except that there is only one outer complex

knockout1(OuterC, InnerComps, UnCovComps) :- !,
        bagof(InC,
              (member(InC, InnerComps),
                 not(covers(OuterC, InC)) ),
            UnCovComps).



% coveredbycomplex(Complex, Events, CoveredE)
%
% Finds all Events which are covered by Complex.  Result is CoveredE.

coveredbycomplex(Complex, Events, CoveredE) :-
        bagof(E, (member(E, Events), covers(Complex, E)), CoveredE), !.
```

```
%                    COMPLEX LEVEL OPERATIONS
%
% A complex is stored as an ordered list of selectors. Selectors are in
% ascending order according to the print name of the attribute.
% In the following code, a form such as "(An = Vn)" represents a selector.



% newselector(Complex, Selector, NewComplex)
%
%  add/replace a selector in a complex, in the correct order.
%
%  Complex    - the original complex
%  Selector   - the selector to be added or replaced
%  NewComplex - the modified complex


newselector([(A1 = V1) | T1], (A1 = V2), [(A1 = V2) | T1]) :- !.
newselector([(A1 = V1) | T1], (A2 = V2), [(A1 = V1) | T3]) :-
        A1 < A2, !,
        newselector(T1, (A2 = V2), T3).
newselector([(A1 = V1) | T1], (A2 = V2), [(A2 = V2), (A1 = V1) | T1]) :-
        A1 > A2, !.
newselector([], Sel, [Sel]) :- !.



% covers(OutComp, InComp)
%
%   Succedes if complex OutComp covers complex InComp, fails otherwise.

covers([(A = OutVal) | OutC], [(A = InVal) | InC]) :- !,
        includes((A = OutVal), (A = InVal)),
        covers(OutC, InC).
covers([(A1 = OutV) | OutC], [(A2 = InV) | InC]) :- !,
        A2 < A1,
        covers([(A1 = OutV) | OutC], InC).
covers([], _) :- !.



% includes(OutSelector, InSelector)
%
%  Succedes if selector OutSelector covers selector InSelector.

includes((Attr = OutVals), (Attr = InVals)) :-
        domaintype(Attr, nominal), !,
        subset(InVals, OutVals).

includes((Attr = OutVals), (Attr = InVals)) :-
        domaintype(Attr, linear), !,
        includeslin(OutVals, InVals).
```

```
includes((Attr = OutVal), (Attr = InVal)) :-
        domaintype(Attr, structured), !,
        ancestor(Attr, OutVal, InVal).



% disjointcomps(Comp1, Comp2)
%
%   Succedes if complexes Comp1 and Comp2 are disjoint (have a null
%   intersection), fails otherwise.

disjointcomps([(A = V1) | T1], [(A = V2) | T2]) :- !,
        disjointsel((A = V1), (A = V2)).
disjointcomps([_ | T1], [_ | T2]) :- !,
        disjointcomps(T1, T2).



% disjointsel(Sel1, Sel2)
%
%   Succedes if selectors Sel1 and Sel2 are disjoint, fails otherwise.

disjointsel((Attr = Vals1), (Attr = Vals2)) :-
        domaintype(Attr, nominal), !,
        disjoint(Vals1, Vals2).

disjointsel((Attr = Vals1), (Attr = Vals2)) :-
        domaintype(Attr, linear), !,
        disjointlin(Vals1, Vals2).

disjointsel((Attr = Val1), (Attr = Val2)) :-
        domaintype(Attr, structured), !,
        not(ancestor(Attr, Val1, Val2)),
        not(ancestor(Attr, Val2, Val1)).



% negate(Complex, NegComps)
%
%   Negate a Complex, returns a list (disjunction) of complexes
%   which cover all points not in the given complex in NegComps.
%   (note: not currently used)

negate(Complex, NegComps) :-
        bagof(NegC,
              (member(Sel, Complex),
                 negatesel(Sel, NSel),
                 NegC = [NSel]),
              NegComps), !.
```

```
% negatesel(Selector, NegSelector)
%
%  Negate an individual Selector, result in NegSelector.

negatesel((Attr = Vals), (Attr = NegVals)) :-
        domaintype(Attr, nominal), !,
        valueset(Attr, AllVals),
        difference(AllVals, Vals, NegVals),
        NegVals /= [].

negatesel((Attr = Vals), (Attr = NegVals)) :-
        domaintype(Attr, linear), !,
        negatelin(Attr, Vals, NegVals),
        NegVals /= [].



% extendagainst(PComp, NegComp, PStar)
%
%  Extend the reference of complex PComp against complex NComp giving
%  a partial star of single selector complexes PStar.


extendagainst([(A = VP) | P], [(A = VN) | N], [[(A = VX)] | X]) :-
        extendref((A = VP), (A = VN), (A = VX)), !,
        extendagainst(P, N, X).
extendagainst([(AP = VP) | P], [(AN = VN) | N], X) :-
        AP < AN, !,
        extendagainst(P, [(AN = VN) | N], X).
extendagainst([(AP = VP) | P], [(AN = VN) | N], X) :- !,
        extendagainst([(AP = VP) | P], N, X).
extendagainst([], _, []) :- !.
extendagainst(_, [], []) :- !.



% extendref(PosSel, NegSel, ExtSel)
%
%   Extend the reference of PosSel against NegSel, giving ExtSel.

extendref((Attr = PosVals), (Attr = NegVals), (Attr = ExtVals)) :-
        domaintype(Attr, nominal), !,
        disjoint(PosVals, NegVals),
        negatesel((Attr = NegVals), (Attr = ExtVals)).

extendref((Attr = PosVals), (Attr = NegVals), (Attr = ExtVals)) :-
        domaintype(Attr, linear), !,
        disjointlin(PosVals, NegVals),
        negatelin(Attr, NegVals, NNVals),
        extendedlin(PosVals, NNVals, ExtVals).
```

```
extendref((Attr = PosVal), (Attr = NegVal), (Attr = ExtVal)) :-
        domaintype(Attr, structured), !,
        ancestor(Attr, ExtVal, PosVal),
        not(ancestor(Attr, ExtVal, NegVal)),
        parent(Attr, ExtVal, ExtParent),
        ancestor(Attr, ExtParent, NegVal).



% refunion(CompList, Refunion)
%
%  Find the refunion of list of complexes "CompList", result is the single
%  complex "Refunion".


refunion([C1, C2 | T], RefU) :- !,
        refu(C1, C2, C3),
        refunion([C3 | T], RefU).
refunion([Comp], Comp) :- !.



% refu(Comp1, Comp2, Refu)
%
%  Finds the refunion of two complexes, Comp1 and Comp2, giving third
%  complex RefU.

refu([(A = V1) | C1], [(A = V2) | C2], [(A = VU) | CU]) :-
        selunion((A = V1), (A = V2), (A = VU)), !,
        refu(C1, C2, CU).
refu([(A1 = V1) | C1], [(A2 = V2) | C2], U) :-
        A1 < A2, !,
        refu(C1, [(A2 = V2) | C2], U).
refu([(A1 = V1) | C1], [(A2 = V2) | C2], U) :- !,
        refu([(A1 = V1) | C1], C2, U).
refu([], _, []) :- !.
refu(_, [], []) :- !.



% selunion(Selector1, Selector2, SelectorUnion)
%
%  Finds the refunion of two selectors, Selector1 and Selector2,
%  giving third selector SelectorUnion.

selunion((Attr = Vals1), (Attr = Vals2), (Attr = UVals)) :-
        domaintype(Attr, nominal), !,
        union(Vals1, Vals2, UVals),
        valueset(Attr, AllVals),
```

```
        not(equals(UVals, AllVals)).    % drop (Attr = *) selectors

selunion((Attr = Vals1), (Attr = Vals2), (Attr = UVals)) :-
        domaintype(Attr, linear), !,
        low(Vals1, L1),
        low(Vals2, L2),
        min([L1, L2], Low),
        highest(Vals1, H1),
        highest(Vals2, H2),
        max([H1, H2], High),
        range(Attr, Min, Max),
        not((Low == Min),(High == Max)),
        UVals = [Low .. High], !.

selunion((Attr = Val1), (Attr = Val2), (Attr = UVal)) :-
        domaintype(Attr, structured), !,
        ancestor(Attr, UVal, Val1),
        ancestor(Attr, UVal, Val2).




% product(Comp1, Comp2, Comp3)
%
%  Finds the product (intersection) of two complexes, Comp1 and Comp2,
%  returns a single complex Comp3.


product([(A = V1) | T1], [(A = V2) | T2], [(A = V3) | T3]) :- !,
        selproduct((A = V1), (A = V2), (A = V3)), !,
        product(T1, T2, T3).
product([(A1 = V1) | T1], [(A2 = V2) | T2], [(A1 = V1) | T3]) :-
        A1 < A2, !,
        product([(A2 = V2) | T2], T1, T3).
product([(A1 = V1) | T1], [(A2 = V2) | T2], [(A2 = V2) | T3]) :- !,
        product([(A1 = V1) | T1], T2, T3).
product(X, [], X) :- !.
product([], X, X) :- !.




% selproduct(Sel1, Sel2, PSel)
%
%  Finds the product (intersection) of two selectors, Sel1 and Sel2,
%  result is selector PSel.

selproduct((Attr = Vals1), (Attr = Vals2), (Attr = ProdVals)) :-
        domaintype(Attr, nominal), !,
        intersection(Vals1, Vals2, ProdVals),
        ProdVals /= [].                 % fail if product is empty
```

```
selproduct((Attr = Vals1), (Attr = Vals2), (Attr = ProdVals)) :-
        domaintype(Attr, linear), !,
        productlin(Vals1, Vals2, ProdVals),
        ProdVals /= [].

selproduct((Attr = Val1), (Attr = Val2), (Attr = Val1)) :-
        domaintype(Attr, structured),
        ancestor(Attr, Val2, Val1), !.
selproduct((Attr = Val1), (Attr = Val2), (Attr = Val2)) :-
        domaintype(Attr, structured),
        ancestor(Attr, Val1, Val2), !.



% trim(Comp, CoveredEvents, TrimmedComp)
%
%  Trim a complex by finding the product of the complex and the refunion
%  of the events covered by the complex.

trim(Comp, CoveredEvents, TrimmedComp) :- !,
        refunion(CoveredEvents, RefU),
        trimcomp(Comp, RefU, TrimmedComp).


% trimcomp(Comp, Refunion, TrimmedComp)

trimcomp([(A = V1) | C1], [(A = VU) | CU], [(A = VT) | CT]) :- !,
        selproduct((A = V1), (A = VU), (A = VT)),
        trimcomp(C1, CU, CT).
trimcomp([(A1 = V1) | C1], [(A2 = VU) | CU], CT) :-
        A2 < A1, !,
        trimcomp([(A1 = V1) | C1], CU, CT).
trimcomp([(A1 = V1) | C1], [(A2 = VU) | CU], [(A1 = V1) | CT]) :- !,
        trimcomp(C1, [(A2 = VU) | CU], CT).
trimcomp(X, [], X) :- !.
trimcomp([], _, []) :- !.
```

```
% Functions for encoding events in a data set .....


% encodeevents(EventList, EncodedEvents)
%
% Encode list of events EventList, returning list of EncodedEvents.

encodeevents([], []) :- !.
encodeevents([E | Rest], [EE | EncodeRest]) :- !,
        encodeevent(E, [], EE),
        encodeevents(Rest, EncodeRest).



% encodeevent(Event, Buff, EncodedEvent)
%
%  .Encode an input event. (i.e. translate to internal form).
%   Events are NOT checked to make sure that values fall within the
%   domain of the attribute.
%
%   Event       - the event in data set format
%   Buff        - Buffer for storing intermediate forms of event
%   EncodedEvent - the event in internal format


encodeevent([], Event, Event) :- !.
encodeevent([Sel | E], PartialEv, NewPartialEv) :- !,
        encodesel(Sel, EncodedSel),
        newselector(PartialEv, EncodedSel, PPEv),
        encodeevent(E, PPEv, NewPartialEv).


% encodesel(Sel, EncodedSel) - Encode a selector.

encodesel([Attr = Val], (Attr = [Val])) :-
        domaintype(Attr, nominal), !.

encodesel([Attr = Val], (Attr = [Val .. Val])) :-
        domaintype(Attr, linear),
        subtype(Attr, integer), !.
encodesel([Attr = Sym], (Attr = [Ord .. Ord])) :-
        domaintype(Attr, linear),
        subtype(Attr, symbolic), !,
        ord(Attr, Sym, Ord).

encodesel([Attr = Val], (Attr = Val)) :-
        domaintype(Attr, structured), !.

encodesel(S, _) :- print("Error - improper selector: ", S).
```

```
% Functions for displaying results .....


% Display the covers of all classes

showcovers :-
        class(Class),
        showcover(Class),
        fail.
showcovers.



% Display the cover of a single class

showcover(Class) :-
        nl, nl,
        print("Cover of class ", Class, ":"), !,
        nl,
        cover(Class, Cover),
        princomplex(Cover),
        nl,
        fail.
showcover(_).




% princomplex(Complex)
%
% Pretty print a complex.

princomplex(Complex) :-
        member(Selector, Complex),
        prinselector(Selector),
        fail.
princomplex(_) :- !.



% prinselector(Selector)
%
% Prettyprint a selector

prinselector((Attr = Vals)) :-
        domaintype(Attr, nominal), !,
        prin("[", Attr, " = "),
        prinlist(Vals),
        prin("]").

prinselector((Attr = Vals)) :-
```

```
        domaintype(Attr, linear),
        subtype(Attr, integer), !,
        prin("[", Attr, " = "),
        prinlin(Vals),
        prin("]").
prinselector((Attr = Vals)) :-
        domaintype(Attr, linear),
        subtype(Attr, symbolic), !,
        prin("[", Attr, " = "),
        prinsym(Attr, Vals),
        prin("]").

prinselector((Attr = Val)) :-
        domaintype(Attr, structured), !,
        prin("[", Attr, " = ", Val, "]").
```

```
% NOMINAL selector RHS functions.
%     The external form of a nominal selector in an input event is
%
%                          [Attr = Val]
%
%     The internal form of a nominal selector in a complex is
%
%                   (Attr = [Val1, Vals2, ... , Valn])
%
%     where each "Val" is a value from the domain of the attribute.
%     The list of values on the right hand side (RHS) of a nominal
%     selector is maintained as an ordered list (with no duplicate Vals).
%
% Associated Declarations:
%     domaintype(Attr, nominal).      - domaintype declaration.
%     valueset(Attr, [list of vals]). - list of all values in the domain.
%


% intersection(X, Y, Z)   the intersection of ordered lists X and Y is Z.

intersection([A | B], [A | C], [A | X]) :- !,
        intersection(B, C, X).
intersection([A | B], [C | D], X) :-
        A < C, !,
        intersection([C | D], B, X).
intersection([A | B], [C | D], X) :- !,
        intersection([A | B], D, X).
intersection(Y, [], []) :- !.
intersection([], Y, []) :- !.




% difference(X, Y, Z)   the difference of ordered lists X and Y is Z.

difference([A | B], [A | C], X) :- !,
        difference(B, C, X).
difference([A | B], [C | D], [A | X]) :-
        A < C, !,
        difference(B, [C | D], X).
difference([A | B], [C | D], [C | X]) :- !,
        difference([A | B], D, X).
difference(Y, [], Y) :- !.
difference([], Y, []) :- !.




% union(X, Y, Z)     the union of ordered lists X and Y is Z.
```

```
union([A | B], [A | C], [A | X]) :- !,
        union(B, C, X).
union([A | B], [C | D], [A | X]) :-
        A < C, !,
        union([C | D], B, X).
union([A | B], [C | D], [C | X]) :- !,
        union([A | B], D, X).
union(Y, [], Y) :- !.
union([], Y, Y) :- !.
```

% disjoint(X, Y)? is true if ordered lists X and Y are disjoint.

```
disjoint([A | B], [C | D]) :-
        A < C, !,
        disjoint([C | D], B).
disjoint([A | B], [C | D]) :-
        C < A, !,
        disjoint([A | B], D).
disjoint(_, []) :- !.
disjoint([], _) :- !.
```

% subset(X, Y)? is true if X is a subset of Y.

```
subset([A | B], [A | C]) :- !,
        subset(B, C).
subset([A | B], [C | D]) :-
        A > C, !,
        subset([A | B], D).
subset([], _) :- !.
```

% equals(X, Y) ? is true if X and Y are equal sets

```
equals(X, Y) :- X = Y, !.
```

% cardinality(X, N) N is the number of elements in set X

```
cardinality(X, N) :- !, length(X, N).
```

```
% LINEAR selector RHS functions.
%     The external form of a linear selector in an input event is
%
%                         [Attr = Val]
%
%     The internal form of a linear selector in a complex is
%
%               (Attr = [L1 .. H1, L2 .. H2, .... , Ln .. Hn])
%
%     where Li and Hi are low and high ends of a segment of values.
%     Limited internal disjuntion is allowed to accomodate "=/=".
%
% Associated Declarations:
%     domaintype(Attr, linear).       - Domaintype declaration.
%     range(Attr, Low, High).         - Low and High limits of domain.
%     order(Attr, [list of symbols]). - Ordered list of elements of domain.
%          (order is used for symbolic valued domains only,
%            the range is computed from the declared order).


% ord(Attr, Sym, N)
%
% Convert ordinal value to symbol and vice-versa for a symbolic linear vbl.
%   Attr  - the attribute name
%   Sym   - symbol name for a given ordinal value
%   N     - ordinal value for a given symbol name

ord(Attr, Sym, N) :-
        order(Attr, L),
        at(Sym, L, N, 1), !.



% at(Symbol, List, OrdVal, 1)
%
%   Associate a Symbol with an ordinal value "OrdVal" in list of
%   symbols "List".

at(Sym, [Sym | X], N, N).
at(Sym, [_ | X], N, I) :-
        J is I + 1,
        at(Sym, X, N, J),!.
at(_ _ _ _) :- print("Error in linear symbol lookup").



% low(SegList, Low) - Get low value from a list of segments.

low([L .. H | _], L).



% highest(SegList, High) - Get highest value from a list of segments
```

```
highest([L .. H], H).
highest([L .. H | X], High) :- !,
        highest(X, High).



% includeslin(OuterSegs, InnerSegs)
%
%  Succedes if list of linear values OuterSegs includes list
%  of linear values InnerSegs, fails otherwise.

includeslin(_, []).
includeslin([Lo .. Ho | Xo], [Li .. Hi | Xi]) :-
        Ho < Li, !,
        includeslin(Xo, [Li .. Hi | Xi]).
includeslin([Lo .. Ho | Xo], [Li .. Hi | Xi]) :- !,
        Lo <= Li,
        Ho >= Hi,
        includeslin([Lo .. Ho | Xo], Xi).



% disjointlin(Segs1, Segs2)
%
%  Succedes if the two lists of linear vals. are disjoint,
%  fails otherwise.

disjointlin([], _) :- !.
disjointlin(_, []) :- !.
disjointlin([L1 .. H1 | X1], [L2 .. H2 | X2]) :-
        H1 < L2, !,
        disjointlin(X1, [L2 .. H2 | X2]).
disjointlin([L1 .. H1 | X1], [L2 .. H2 | X2]) :-
        H2 < L1, !,
        disjointlin(X2, [L1 .. H1 | X1]).



% negatelin(Attribute, PosLinVals, NegLinVals)
%
%   Negate a list of linear vals. PosLinVals giving NegLinVals
%   for given Attribute.

negatelin(Attr, [LP .. HP | XP], N) :- !,
        neglinlow(Attr, LP, Low),
        neglinmid([LP .. HP | XP], HI, Mid),
        neglinhi(Attr, HI, High),
```

```
        appendx([Low, Mid, High], N).

neglinlow(Attr, LP, []) :-
        range(Attr, LP, _), !.
neglinlow(Attr, LP, [Low .. H]) :- !,
        range(Attr, Low, _),
        H is LP - 1.

neglinmid([L .. H], H, []) :- !.
neglinmid([L1 .. H1, L2 .. H2 | X], HI, [L .. H, N]) :-
        L2 > H1 + 1, !,
        L is H1 + 1,
        H is L2 - 1,
        neglinmid([L2 .. H2 | X], HI, N).
neglinmid([L1 .. H1, L2 .. H2 | X], HI, N) :- !,
        neglinmid(X, HI, N).

neglinhi(Attr, HI, []) :-
        range(Attr, _, HI), !.
neglinhi(Attr, HI, [L .. High]) :- !,
        range(Attr, _, High),
        L is HI + 1.




% extendedlin(PVals, NVals, XVals)
%
%  Collect segments from second arg. which are extensions of segments
%  in first arg. and return in third arg.

extendedlin(_, [], []) :- !.
extendedlin([], _, []) :- !.
extendedlin([LP .. HP | XP], [LN .. HN | XN], XVals) :-
        HN < LP, !,
        extendedlin([LP .. HP | XP], XN, XVals).
extendedlin([LP .. HP | XP], [LN .. HN | XN], XVals) :-
        HP < LN, !,
        extendedlin(XP, [LN .. HN | XN], XVals).
extendedlin([LP .. HP | XP], [LN .. HN | XN], [LN .. HN | XVals]) :- !,
        LN <= LP,
        HN >= HP,
        extendedlin(XP, XN, XVals).




% productlin(Vals1, Vals2, ValsProd)
%
%  Find the product (intersection) of two lists of linear vals,
%  Vals1 and Vals2, giving ValsProd.
```

```
productlin([], _, []) :- !.
productlin(_, [], []) :- !.
productlin([L1 .. H1 | X1], [L2 .. H2 | X2], P) :-
        H1 < L2, !,
        productlin(X1, [L2 .. H2 | X2], P).
productlin([L1 .. H1 | X1], [L2 .. H2 | X2], P) :-
        H2 < L1, !,
        productlin(X2, [L1 .. H1 | X1], P).
productlin([L1 .. H1 | X1], [L2 .. H2 | X2], [L .. H | P]) :- !,
        max([L1, L2], L),
        min([H1, H2], H),
        productlin(X1, X2, P).
```

```
% prinlin(Lin_Vals) - Print a list of integer linear values

prinlin([A]) :- !,
        prinseg(A).
prinlin([A, B]) :- !,
        prinseg(A),
        prin(" v "),
        prinlin(B).
prinlin([]) :- !,
        print("Error - null RHS in linear selector.").

prinseg(L .. H) :-
        L = H,
        prin(L), !.
prinseg(A) :-
        prin(A), !.
```

```
% prinsym(Attr, Lin_Vals)
%
% Print a list of symbolic linear values for a given attribute

prinsym(Attr, [A]) :- !,
        prinsymseg(Attr, A).
prinsym(Attr, [A, B]) :- !,
        prinsymseg(Attr, A),
        prin(", "),
        prinsym(Attr, B).
prinsym(_, []) :- !, print("Error - null RHS in linear selector.").

prinsymseg(Attr, L .. H) :-
        L = H,
        ord(Attr, Sym, L),
        prin(Sym), !.
```

```
prinsymseg(Attr, L .. H) :-
        ord(Attr, SymL, L),
        ord(Attr, SymH, H),
        prin(SymL .. SymH), !.
```

```
% STRUCTURED selector RHS functions.
%     The internal form of a structured selector is (Attr = Val).
%     No internal disjunction is allowed.
%     A node may have only one parent in the abstraction hierarchy.
%
%     The external form of a structured selector in an input events is
%
%                         [Attr = Val]
%
%     The internal form of a structured selector in a complex is
%
%                         (Attr = Val)
%
% Associated declarations:
%     domaintype(Attr, structured).          - domaintype de  ration.
%     structure(Attr,[parent([Sibs],Parent),...])
%                                                - structure specification
%
% Additional associated structures:
%     ancestry(Attr, Node, [ancestor list]).  - list of ancestors for
%               each node in the structure, computed from structure spec.
%



% ancestor(Attribute, HiNode, LoNode)
%
%  Succedes if HiNode is an ancestor of LoNode for given Attribute,
%  fails otherwise. A node may be an ancestor of itself.

ancestor(Attr, X, X).
ancestor(Attr, HiNode, LoNode) :-
        ancestry(Attr, LoNode, Alist),
        member(HiNode, Alist).



% parent(Attr, Node, Parent)
%
%  Succedes if Parent is a parent of Node for given Attribute, fails
%  otherwise. Can also be used to generate all parents of a Node for
%  a given Attribute.

parent(Attr, Node, Parent) :-
        ancestry(Attr, Node, [P |_]).



%
% Structured domain initialization .....
%
```

```
% explodestruc(Attr, StructureSpec)
%
% Explode a structure specification into a set of ancestry lists, one for
% each node in the structure. An ancestry list is a list of all ancestors,
% in order, for a given node. Attr is an attribute, StructureSpec is a
% structure specification from a data set.

explodestruc(Attr, StructureSpec) :-
        allnodes(StructureSpec, NodeList),
        member(Node, NodeList),
        ancestorlist(Node, StructureSpec, AList),
        assert(ancestry(Attr, Node, AList)),
        fail.
explodestruc(_, _).



% allnodes(StructureSpec, NodeList)
%
%   Given structure specification StructureSpec, make a list of all
%   nodes in a structured domain, return in NodeList.

allnodes([], []) :- !.
allnodes([parent(Sibs, P) | X], NodeList) :- !,
        sort(Sibs, L1),
        union(L1, [P], L2),
        allnodes(X, L3),
        union(L2, L3, NodeList).



% ancestorlist(Node, StructureSpec, Ancestry)
%
%   Given a Node and a structure specification StructureSpec, make an
%   ancestor list for a node in a structured domain, return in Ancestry.

ancestorlist(Node, StructureSpec, [P | X]) :-
        father(Node, StructureSpec, P), !,
        ancestorlist(P, StructureSpec, X).
ancestorlist(_, _, []) :- !.



% father(Node, StructureSpec, Parent)
%
%   Given a Node and a structure specification StructureSpec, find the
%   father of the node (fail if none found)

father(Node, [parent(Sibs, P) | X], P) :-
        member(Node, Sibs), !.
father(Node, [_ | X], P) :-
        father(Node, X, P), !.
```

```
%
%                    BASIC LIST OPERATIONS
%

% first(List, A)   A is the first element of List

first([A | B], A) :- !.



% member(X, Y)?  is true if X is a member of list Y.

member(X, [X | _]).
member(X, [_ | Y]) :-  member(X, Y).


% append(X, Y, Z)    the result of concatenating lists X and Y is Z.

append([], Y, Y) :- !.
append([A | B], Y, [A | C]) :- !, append(B, Y, C).


% appendx(X, Y)      Y is the list of sublists in X appended together

appendx(X, Y) :-  bagof(A, (member(B, X), member(A, B)), Y), !.


% firstn(X, N, Y)    Y is the first N elements of X

firstn(_, 0, []) :-  !.
firstn([A | B], N, [A | C]) :- !,
        M is N-1,
        firstn(B, M, C).
firstn([], _, []) :- !.


% following(X, List, AfterX)  AfterX is all elements in List following X

following(X, [X | AfterX], AfterX) :- !.
following(X, [_ | List], AfterX) :- !,
        following(X, List, AfterX).
following(X, [], []) :- !.
```

```
% sort(L0, L)  L is the result of sorting L0 into ascending order
%             quicksort algorithm - from prolog lib.

sort(L0, L) :-  qsort(L0, L , []), !.

qsort([X | L], R, R0) :- !,
        partition(L, X, L0, L1),
        qsort(L1, R1, R0),
        qsort(L0, R, [X | R1]).
qsort([], R, R) :- !.



% partition(L, Pivot, Low, High)  partition L into Low and High
%                                 sublists around the given pivot.

partition([X | L], Y, [X | L0], L1) :-
        X <= Y, !,
        partition(L, Y, L0, L1).
partition([X | L], Y, L0, [X | L1]) :- !,
        partition(L, Y, L0, L1).
partition([], _, [], []).



% remove(X, Y, Z)  Z is list Y with all occurrences of X removed

remove(X, [], []) :- !.
remove(X, [X | B], C) :- !, remove(X, B, C).
remove(X, [A | B], [A | C]) :- !, remove(X, B, C).



% prinlist(List)  prin a list of items separated by "v"'s

prinlist([A]) :- prin(A), !.
prinlist([A | B]) :- !, prin(A, " v "), prinlist(B).
prinlist([]) :- print("What?"), !.



% min(X, M)  M is the minimum of list of numbers X

min([X, Y | T], Z) :-
        X <= Y, !,
        min([X | T], Z).
min([X, Y | T], Z) :- !,
        min([Y | T], Z).
```

```
min([X], X) :- !.


% max(X, M)  M is the maximum of list of numbers X

max([X, Y | T], Z) :-
        X >= Y, !,
        max([X | T], Z).
max([X, Y | T], Z) :- !,
        max([Y | T], Z).
max([X], X) :- !.


% Other auxiliary functions:

% Time how long it takes to evaluate goal "P"

time(P) :-
        cputime(Start),
        P,
        cputime(Finish),
        Total is Finish - Start,
        Nsecs is Total / 60,
        LeftOver is Total mod 60 * 100 / 60,
        (LeftOver < 10 -> Sep = '.0' | Sep = '.'),
        print('CPU Time = ', Nsecs, Sep, LeftOver, ' secs').


% Signon messages:


nl, nl, print("Aq-Prolog [Version 1.0|"),
nl,     print("Type 'data(<filename>)!' to load a data file"),
        print("Type 'run!' to start Aq"), nl!
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-F-85-930	2.	3. Recipient's Accession No.

4. Title and Subtitle

AQ-PROLOG: A Prolog Implementation of an Attribute-Based Inductive Learning System

5. Report Date January 1985

6.

7. Author(s)

Jeffrey M. Becker

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Department of Computer Science
University of Illinois
Urbana, IL 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

ONR N00014-82-K-0186
NSF DCR 84-06801

12. Sponsoring Organization Name and Address

Office of Naval Research, Arlington, VA
National Science Foundation, Washington, DC

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This paper describes the Aq-Prolog program for learning discriminant descriptions of classes of objects from examples. It also describes the use of the program, and provides an evaluation of the program and the implementation task. This project is a test of the suitability of Prolog as an implementation language for programs based on the Aq.

17. Key Words and Document Analysis. 17a. Descriptors

Machine Learning
Discrimination
Prolog
Interlisp
Inductive Inference Decision Rules

17b. Identifiers/Open-Ended Terms

AQ-PROLOG, VL1

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 64
	20. Security Class (This Page) UNCLASSIFIED	22. Price