Generating Declarative Language Bias for Top-Down ILP Algorithms

Stefan Kramer
Austrian Research Institute for Artificial Intelligence
Schottengasse 3
A-1010 Vienna, Austria
stefan@ai.univie.ac.at

Abstract

Many of today's algorithms for Inductive Logic Programming (ILP) put a heavy burden and responsibility on the user, because their declarative bias have to be defined in a rather low-level fashion. To address this issue, we developed a method for generating declarative language bias for top-down ILP systems from high-level declarations. The key feature of our approach is the distinction between a user level and an expert level of language bias declarations. The expert provides abstract meta-declarations, and the user declares the relationship between the meta-level and the given database to obtain a low-level declarative language bias. The suggested languages allow for compact and abstract specifications of the declarative language bias for top-down ILP systems using schemata. We verified several properties of the translation algorithm that generates schemata, and applied it successfully to a few chemical domains. As a consequence, we propose to use a two-level approach to generate declarative language bias.

1 Introduction

Many of today's Inductive Logic Programming (ILP) algorithms put a heavy burden and responsibility on the user, because their declarative language biases have to be defined in a rather low-level fashion. While all sorts of declarative biases have been proposed in the past (e.g., [21, 1], or [17] for an excellent overview) and the things that could be specified are relatively clear, the level of abstraction of current formal languages leaves much to be desired.

As a consequence, ILP algorithms can currently only be applied by ILP experts.[1] It might also be the reason why apparently ILP algorithms have not yet been adopted by tool vendors offering tools for *Knowledge Discovery and Data Mining (KDD)*. We thus think that one way to improve the acceptance

[1]There have been, nevertheless, several successful applications of ILP algorithms to real-world problems, e.g., [20, 10, 13, 15].

could be to devise easy-to-use methods for the specification of declarative language bias for relational application domains.

In particular, generating or compiling a low-level specification from an abstract high-level specification appears to be an option. In fact, this has been hinted at in [4] and in [14]. In this paper we propose a method for generating the declarative language bias of top-down ILP algorithms based on schemata.[2] Schemata as used in several ILP algorithms (e.g., FOCL [19], FOSSIL [12], SRT [14], and TILDE [2]) are a comparably useful and practical form of declarative bias for ILP, but their definition is still a hard and error-prone task.

The suggested solution amounts to a two-level approach, where the user just has to declare the relationship between the meta-level and the given database properly, and the expert has to declare the meta-level information (consisting of the available meta-literals and the meta-schemata). From these two kinds of declarations, schemata for top-down ILP systems can be generated.

The main benefit of this approach is that more abstract and compact declarations are possible. This can be compared with the benefits of abstraction in high-level programming languages. High-level statements serve as a shorthand for more complex statements, they are easier to specify and easier to understand.

The first part of this paper will illustrate the basic ideas, mostly by means of examples. The definition of the syntax of the suggested language can be found in appendix A. The algorithm generating schemata is described in appendix B.

In the next section, we briefly describe the use of schemata as declarative language bias. In section 3, we present the two-level approach to generating schemata from meta-declarations. Section 4 deals with the verification and validation of our approach. In section 5, we discuss related work on declarative language bias.

2 Schemata as Declarative Bias

Several ILP systems, such as FOCL, FOSSIL, SRT and TILDE use schemata as declarative bias. The idea of schemata can easily be explained using an example:

```
schema((sym_bond(A,B,C,D), atm(A,C,E,F,G), G>H),
       [A:chemical:'+',  B:atomid:'+',   C:atomid:'-',
        D:bondtype:'-',  E:element:'-',   F:atomtype:'-',
        G:charge:'-',    H:charge:'=']).
```

This expression specifies an admissible refinement of a given clause by the addition of a conjunction of literals (`sym_bond(A,B,C,D), atm(A,C,E,F,G), G>H`). The first subexpression can also consist of a single condition only. The refinement is constrained by the specifications in the second subexpression, a

[2]We employ the term "schemata" for specifications of admissible refinements of clauses at the first-order level. In contrast to this, rule models [16], also sometimes referred to as rule schemata, are second-order declarations of complete rules.

list containing type and mode declarations of the variables. The variables in the list refer to argument positions in the conjunction of literals. This kind of reference to argument positions is used throughout the paper, both on the "user level" and on the "expert level". All variables occurring in the literals have to occur in the second subexpression, and vice versa.

If a given clause is to be refined by means of such a schema, the variables labeled as '+' are unified with variables already bound, provided that the types are matching as well. If a variable is labeled as '–', it means that this variable is not yet bound before the application of the schema. A new variable can be used in subsequent applications of the schemata, given that the mode is '+' there and the types are matching. So, in the example, variable `C` can be used in any subsequent literal, if all constraints are fulfilled. The mode declaration '=' means that a constant will be inserted at the respective argument position.

In practice, schemata are a comparably convenient form of specifying a declarative language bias. Their declaration is nevertheless a burdensome task, and the declared schemata tend to look very similar for many application domains (e.g., for chemical domains). So, generating schemata from abstract high-level declarations appears to be possible as well as desirable.

3 Generating Schemata from Meta-Schemata: A Two-Level Approach

In this section we present a two-level approach to generating schemata from what we call "meta-schemata". The approach distinguishes two levels of declarations, namely the "expert level", and the "user level". The expert level consists of high-level declarations (e.g., the meta-schemata), and the user level consists of declarations to relate the given relations to the expert level.

For convenience, we will also make use of the common distinction between "meta" and "object": the second-order declarations will be said to be on the "meta-level", and the first-order declarations to be on the "object-level".

The goal in the design of the languages was to keep the declarations to be made by the user as simple as possible, and to keep the languages for both levels as similar as possible. There is a correspondence between the object-level and the meta-level: at the object-level, there are variables, types, literals, and schemata, and at the meta-level, there are meta-variables, meta-types, meta-literals and meta-schemata. This correspondence is intended to increase the ease of use.

In the following subsections, we will use a running example to illustrate the generation of schemata. In the example, the meta-schemata are defined for labeled graphs, and the user declarations are made to match these graph-based meta-schemata with a given chemical database (containing descriptions of molecules at the atom- and bond level).

3.1 The Expert Level

This subsection describes the expert-level declarations. First of all, the literals that can be used in the declarations of the meta-schemata are stated. In the following, the literals at the meta-level will be called "meta-literals", the variables of the meta-literals "meta-variables" and the types of meta-variables "meta-types". In our approach, meta-variables denote sets of variables at the "object"-level.

This is an example of a declaration of a meta-literal:

```
meta_literal(arc_relation(A,B,C,D,E,F),
             [A:example:'1-1',
              B:node:'1-1', C:node:'1-1',
              D:sym_arc_feature:'0-n',
              E:num_arc_feature:'0-n',
              F:other_arc_arg:'0-n']).

meta_literal(node_relation(A,B,C,D,E),
             [A:example:'1-1',
              B:node:'1-1',
              C:sym_node_feature:'0-n',
              D:num_node_feature:'0-n',
              E:other_node_arg:'0-n']).
```

These statements declare that `node_relation` has five arguments, and that `node_relation` has six arguments. The second subexpression in such a declaration assigns meta-types (e.g., `node`, `sym_node_feature` meaning a symbolic property of a node, and `num_arc_feature` meaning a numeric property of an arc) and cardinality constraints (`1-1, 0-1, 1-n` or `0-n`) to meta-variables.

Cardinality constraints refer to the sets of variables assigned to a meta-variable. They provide upper and lower bounds for the admissible number of assignments of variables to a meta-variable.

Note that in the declaration of meta-literals (and of meta-schemata) the order of the meta-variables is irrelevant. The order of variables is relevant only at the "object"-level, when conjunctions of literals are variabilized during the construction of schemata.

Such meta-literals are used in the declarations of meta-schemata. The declaration of meta-literals has to be consistent with the usage in the meta-schemata (with respect to cardinality constraints and meta-types). All meta-literals used in meta-schemata have to be declared before.

The following expression is an example for a specification of a meta-schema:

```
meta_schema((arc_relation(A,B,C,D,E,F), node_relation(A,C,G,H,I,J)),
            [A:example:'1-1':'+', B:node:'1-1':'+', C:node:'1-1':'-',
             D:sym_arc_feature:'0-n':'-', E:num_arc_feature:'0-n':'-',
             F:other_arc_arg:'0-n':'-',   G:sym_node_feature:'1-1':'=',
             H:sym_node_feature:'0-n':'-',I:num_node_feature:'0-n':'-',
             J:other_node_arg:'0-n':'-']).
```

Such a declaration consists of the declaration of a conjunction of meta-literals, and meta-types, cardinality constraints and modes for the meta-variables used.

Note that in the example meta-variables of meta-type `sym_node_feature` occur twice. For one of them, a single variable of mode '=' will occur in the generated schema(ta), and for the other, an arbitrary number of variables of mode '–' may occur.

The cardinality constraints in such a declaration have to be consistent with the declarations of the meta-literals so that the sum of the cardinalities of meta-variables of the same meta-type used in the same meta-literal is within the range specified in the meta-literal declaration (i.e., it should not violate the cardinality constraint defined there).

In general, meta-schemata should best be read as conditionals: if there is a match with declarations at the object-level, a corresponding schema is generated.

3.2 The User Level

This subsection describes the declarations to be made by the user. One of the main differences between the expert level and the user level is that at the user level there are no cardinality constraints.

Firstly, the user has to declare the relations that are to be considered in the translation. This declaration includes the types of variables, since they are used in the generated schemata:

```
relation(atm(A,B,C,D,E),
         [A:chemical, B:atomid, C:element,
          D:atomtype, E:charge]).

relation(sym_bond(A,B,C,D),
         [A:chemical, B:atomid, C:atomid,
          D:bondtype]).
```

Obviously, this way of labeling argument positions could be made simpler, but our aim was to keep the different sorts of declarations as similar as possible.

Secondly, the connection between the relations of the domain and the meta-level has to be declared. Here, each relation is declared to be of some meta-relation, and the argument positions are labeled with meta-types:

```
node_relation(atm(A,B,C,D,E),
              [A:example, B:node,  C:sym_node_feature,
               D:sym_node_feature, E:num_node_feature]).

arc_relation(sym_bond(A,B,C,D),
              [A:example, B:node, C:node,
               D:sym_arc_feature]).
```

The first declaration states that relation **atm** contains information about properties of nodes according to the graph-based declarations at the meta-level. It says, e.g., that variable **B** of type **atomid** is of meta-type **node**, and that variable **E** of type **charge** is of meta-type **num_node_feature**. The second declaration states that relation **sym_bond** "contains" the arcs of the graphs as well as their properties.

At the user level, meta-types are merely used for labeling the argument positions. So, meta-types have a different meaning at the user level.

Note that at this level only a subset of all potential meta-variables of some meta-type declared at the expert level has to occur. The declaration at the user level, however, has to be consistent with the lower bounds of the cardinality constraints defined there.

The above declarations are sufficient to generate a schema based on them. The output generated in our example looks like this:

```
schema((sym_bond(A,B,C,D),atm(A,C,E,F,G)),
       [A:chemical:'+', B:atomid:'+',   C:atomid:'-',
        D:bondtype:'-', E:element:'=',  F:atomtype:'-',
        G:charge:'-']).
```

A translation algorithm generating such schemata can be found in appendix B. One of the main tasks of the algorithm is the assignment of variables to meta-variables, which is heavily constrained by types, meta-types and cardinality constraints. We employ a simple generate-and test algorithm for this task. A more sophisticated constraint satisfaction algorithm would make more informed choices from the beginning and thus avoid unnecessary backtracking.

4 Verification and Validation of the Approach

In appendix C, we verify some of the properties of the translation algorithm. Most importantly, it can be shown that the cardinality constraints are never violated by the algorithm. Secondly, the algorithm, by design, always fulfills the type constraints for the assignment of variables. This also holds with respect to meta-types.

We also did some initial validation of the approach: the algorithm has been applied to several chemical domains ([9], [20], [13]) using all graph-based meta-schemata (see appendix D). Subsequently, we compared the results generated by the algorithm with the manually engineered declarative language bias. Not surprisingly, there were only a few observed differences. Some of the generated schemata were not included in the manually engineered declarations, mostly for efficiency reasons.

The ultimate validation of such an approach are usability tests with real users, which will show the utility of the approach. Such tests checking the user-friendliness of the suggested language are under way.

5 Related Work

Closely related approaches in the literature can be described along two dimensions: firstly, there are languages for the specification of complete rules, and languages for the specification of refinements only. Secondly, these declarations can be at the meta-level or at the object-level. Using these two dimensions, various existing systems and languages can be categorized as follows[3]:

1. **Specification of complete clauses, meta-level:**
   Rule schemata [11], rule models [16], second-order schemata [6]

2. **Specification of refinement, meta-level:**
   Relational clichés [19, 18]

3. **Specification of refinement, object-level:**
   FOSSIL [12], SRT [14], TILDE [2]

The approach presented here belongs to the second group of methods, as it allows for the specification of refinements at the meta-level. Next, we will compare it with probably the closest work in the literature, the approach based on relational clichés. Subsequently, we will compare it with two other well-known techniques for specifying declarative language bias, ADGs [4] and Dlab [7].

5.1 Relational Clichés

In [19], a meta-level approach to look-ahead for top-down ILP systems is presented. The look-ahead is based on so-called relational clichés, which are second-order expressions defining admissible refinements of a clause. If there is a match with predicates at the object-level, the cliché is applied. The subsequent expressions are examples of relational clichés as defined in [18]:

```
Pattern:    part-of(A,B) & ext-pred(...,B,...)
var restr:  {[introduces-new-var   non-numeric 2],
             [include-old-variable non-numeric 1]} &
            {[include-new-var      non-numeric *ANY*]}
pred restr: {[pred-type = ext-pred, include-pred(part-of)]} &
```

[3]This list contains just a small fraction of the approaches described in the literature.

```
                {[pred-type = ext-pred]}

    Pattern:    ext-pred(...,A,...) & thresh-comp(A, Thresh)
    var restr:  {[introduces-new-var   numeric *ANY*]} &
                {[include-new-var      numeric 1]}
    pred restr: {[pred-type = ext-pred} &
                {[pred-type = thresh]}
```

In the "variable restrictions" (`var restr`), the second term denotes the type of the variable, and the third one the position of the variable in the respective literal. The following meta-schemata express exactly the same thing:

```
meta_schema((part_of(A,B), ext_pred(B,C)),
            [A:non_numeric:'1-1':'+',
             B:non_numeric:'1-1':'-',
             C:other:'0-n':'-']).

meta_schema((ext_pred(A, B), thresh_comp(A, Thresh)),
            [A:numeric:'1-1':'+',
             B:other:'0-n':'+',
             Thresh:numeric:'1-1':'=']).
```

The main difference between our approach and [19] is that we propose a two-level, generative approach, and that in [19] clichés are applied "on the fly". We think there are several advantages of our approach over [19]. Firstly, it allows for a degree of user-control that cannot be accomplished with a "dynamic" approach. The user can inspect the generated schemata, and delete some of them, if needed, to restrict the search space. Secondly, it is more transparent in that the users cannot mix up the levels. Thirdly, the same schemata are repeatedly used during learning. Thus, it is not necessary to derive them anew every time they are applied.

Besides, the representation developed in [18] (see the example above) is not as expressive as the language proposed in this paper. The representation proposed there is especially restricted with respect to variables.

5.2 ADGs and Augmented ADGs

ADGs [4] are grammars for the declaration of antecedents of rules. In ADGs, variables are logical variables. Variables in a right-hand side of a grammar rule that are not in the left-hand side are distinct for each application of the rule. So, generating new variables is easy with ADGs. The hard task is to keep track of them so that they can be reused [5].

To illustrate the reuse of variables with ADGs, we present an example that is first formulated in terms of meta-schemata and schemata, and then in terms of ADGs. For completeness, we include all declarations that are used for the generation of schemata.

```
/* P A R T - O F   E X A M P L E */

/* EXPERT LEVEL */

/* meta-literals */

meta_literal(part_of(A,B),
             [A:non_numeric:'1-1',
              B:non_numeric:'1-1']).

meta_literal(ext_pred(B,C),
             [B:non_numeric:'1-1',
              C:other:'0-n']).

/* meta-schemata */

meta_schema((part_of(A,B), ext_pred(B,C)),
            [A:non_numeric:'1-1':'+',
             B:non_numeric:'1-1':'-',
             C:other:'0-n':'-']).

meta_schema(ext_pred(B,C),
            [B:non_numeric:'1-1':'+',
             C:other:'0-n':'-']).

/* USER LEVEL */

/* object-level declarations */

relation(part_of(A, B), [A:object, B:object]).

relation(test1(A), [A:object]).
relation(test2(A), [A:object]).

/* meta-level declarations */

part_of(part_of(A, B),  [A:non_numeric, B:non_numeric]).

ext_pred(test1(A), [A:non_numeric]).
ext_pred(test2(A), [A:non_numeric]).

/* generated schemata */

schema((part_of(A,B),test1(B)), [A:object:'+',B:object:'-']).
schema((part_of(A,B),test2(B)), [A:object:'+',B:object:'-']).
schema((part_of(A,B),test1(A)), [A:object:'-',B:object:'+']).
schema((part_of(A,B),test2(A)), [A:object:'-',B:object:'+']).
schema(test1(A), [A:object:'+']).
schema(test2(A), [A:object:'+']).
```

The following ADG defines the same language bias:

```
            body(X) --> schemata(X).
            schemata(X) --> schema(X),schemata(X).
            schemata(X) --> [].
            schema(X) --> ext_pred(X).
            schema(X) --> [part_of(X,Y)],ext_pred(Y),schemata(Y).
            ext_pred(Z) --> [test1(Z)].
            ext_pred(Z) --> [test2(Z)].
```

As can be seen in the example, reuse works by a kind of recursion. This is of course problematic, when more than one variable needs to be used for subsequent variabilizations of literals. This is shown in the following example:

```
/* EXPERT LEVEL */

/* meta-literals */

meta_literal(mp(X,Y),
             [X:non_numeric:'1-1',
              Y:non_numeric:'1-1']).

/* meta-schemata */

meta_schema(mp(X, Y),
            [X:non_numeric:'1-1':'+', Y:non_numeric:'1-1':'+']).

meta_schema(mp(X, Y),
            [X:non_numeric:'1-1':'+', Y:non_numeric:'1-1':'-']).

/* USER LEVEL */

/* object-level declarations */

relation(p(X, Y),     [X:object, Y:object]).
relation(q(X, Y),     [X:object, Y:object]).

/* meta-level declarations */

mp(p(X,Y), [X:non_numeric, Y:non_numeric]).
mp(q(X,Y), [X:non_numeric, Y:non_numeric]).

/* generated schemata */
```

```
schema(p(A,B), [A:object:'+',B:object:'+']).
schema(q(A,B), [A:object:'+',B:object:'+']).
schema(p(A,B), [A:object:'+',B:object:'-']).
schema(p(A,B), [A:object:'-',B:object:'+']).
schema(q(A,B), [A:object:'+',B:object:'-']).
schema(q(A,B), [A:object:'-',B:object:'+']).
```

Given this specification, clauses like this will be generated:

```
  h(X) :- p(X Y), q(X, Z).
  h(X) :- p(X,X), p(X, Y), q(Y,Z), p(Z, X).
  ...
```

Since binary literals need to be variabilized, and the variablizations depend on the variables "introduced" so far, ADGs cannot express this in a compact form.

Augmented ADGs [3], however, provide a mechanism for reusing variables in a flexible way. Among other things, Augmented ADGs include meta-variables, which can be used to denote sets of variables. So, meta-variables can serve the same purpose as in our approach.

One advantage of schemata over ADGs is that one does not have to specify a "global" grammar, but only "local" schemata used for refinement. Admissible variabilizations might be easier to declare using types and modes than using ADGs [4].

5.3 Dlab

A Dlab [7] grammar is a declaration of all literals that can be used in a clause using sets of literals, including some cardinality constraints on the number of literals to be selected from these sets of literals.

The number of variables in a Dlab grammar is restricted. So, all variables to be used have to be specified in such a grammar. There is no way of introducing an arbitrary number of new variables as with ADGs or with schemata.

To allow for more compact declarations, meta-variables can be used as well, but they only denote function symbols and predicate symbols, not variables.

Consequently, Dlab does not offer a flexible way of expressing various variabilizations of a literal – the user has to enumerate all possibilities. Variables introduced in a literal cannot be handled elegantly in subsequent literals.

This is illustrated using our example from above, this time as a Dlab grammar. The example shows only variabilizations for variables `X` and `Y`, as more variables would complicate the declaration considerably. In the example, all "dependent" variabilizations have to be enumerated.

```
example_temps = {h(X) <---
   1-2:[mp(X,X), 0-1:[mp(X,Y),
                      len-len:[mp(X,Y), mp(Y,Y)]]
```

```
example_vars = {
  dlab_variable{mp, 1-1, [p, q]}
}
```

However, it is also possible to specify type and mode information in Dlab. These declarations are checked after the generation of a clause. In this sense, they are not embedded in Dlab itself, but act as a filter on Dlab's output. For efficiency, this kind of information should be encoded in the language templates whenever possible, because then unwanted clauses are never generated [8].

One of the main differences between Dlab and schemata is that schemata are usually a lot easier to declare, but they also produce a lot more unwanted clauses. This does not pose a serious problem to greedy algorithms such as SRT, but probably to algorithms inducing interesting patterns (e.g., Claudien). In the latter case, the algorithms have to minimize the number of queries for efficiency reasons [8]. To reduce the number of queries also in the former case, SRT offers the possibility to check the output generated by schemata using arbitrary conditions that can be specified by the user.

5.4 Differences between Meta-Declarations in Augmented ADGs, Dlab and Meta-Schemata

In contrast to Augmented ADGs and Dlab, the meta-level and the object-level in our approach (by design) look very much the same. Meta-level declarations in Augmented ADGs and in Dlab have a kind of macro flavor. One technical difference is that in our approach meta-variables have types and modes. The advantages of types and modes at the meta-level are the same as those at the object-level – they allow for very compact and abstract declarations.

6 Conclusion and Further Work

In essence, this work shows that declarative bias for top-down ILP systems can be generated from high-level declarations specified by the user. Meta-declarations have been used for declarative language bias before, but they have not yet been used to generate schemata. Of course, schemata cannot be generated out of nothing: some high-level specification has to be available. In our case, meta-schemata serve this purpose.

The suggested languages allow for a compact and abstract specification of the declarative language bias of top-down ILP systems. The key feature of our approach is the distinction between the user level and the expert level, which roughly corresponds to the distinction between the "object"-level and the meta-level. This distinction is intended to make the bias declaration task easier for the user. If the idea of such a separation turns out to be useful in practice, it might contribute to a wider usage of ILP algorithms.

Apart from the suggested distinction between a user level and an expert level, our work offers a number of technical novelties, such as types, modes, and cardinality constraints of meta-variables.

One of the limitations is that this work is applicable only to top-down ILP algorithms. However, schemata and the idea of generating schemata might be even wider applicable. For instance, the most specific clause constructed by Progol could be sorted according to schemata in order to enable more efficient proofs.

Acknowledgements

This research is partly sponsored by the project "Carcinogenicity Detection by Machine Learning", supported by the Austrian Federal Ministry of Science and Transport. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry of Science and Transport. I would like to thank W.W. Cohen, L. Dehaspe, J. Fürnkranz, M. Pazzani, B. Pfahringer and G. Widmer for valuable discussions.

References

[1] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In R. Bajcsy, editor, *Proc. Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 1044–1049, San Mateo, CA, 1993. Morgan Kaufmann.

[2] H. Blockeel and L. De Raedt. Top-down induction of logical decision trees. Technical Report Report CW 247, Katholieke Universiteit Leuven, Belgium, 1997.

[3] W.W. Cohen. Rapid prototyping of ilp systems using explicit bias. In F. Bergadano, editor, *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, Chambery, France, 1993.

[4] W.W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2), 1994.

[5] W.W. Cohen, 1998. Personal Communication.

[6] L. De Raedt and M. Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8(2), 1992.

[7] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.

[8] L. Dehaspe, 1998. Personal Communication.

[9] S. Džeroski and B. Kompare, 1995. Personal Communication.

[10] S. Džeroski, S. Schulze-Kremer, K. Heidtke, K. Siems, and D. Wettschereck. Applying ilp to diterpene structure elucidation from ^{13}c nmr spectra. In S. Muggleton, editor, *Proceedings of the 6th International Workshop on Inductive Logic Programming (ILP-96)*, Berlin Heidelberg New York, 1996. Springer.

[11] W. Emde, C.U. Habel, and C.-R. Rollinger. The discovery of the equator or concept-driven learning. In *Proc. International Joint Conference on Artificial Intelligence 1983*, 1983.

[12] J. Fürnkranz. *Efficient Pruning Methods for Relational Learning.* PhD thesis, Vienna University of Technology, Vienna, Austria, 1994.

[13] R.D. King and A. Srinivasan. Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental Health Perspectives*, 1997.

[14] S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[15] S. Kramer, B. Pfahringer, and C. Helma. Mining for causes of cancer: Machine learning experiments at various levels of detail. In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97)*, Menlo Park, CA, 1997. AAAI Press.

[16] K. Morik. Balanced cooperative modeling. *Machine Learning*, 11(2-3), 1993.

[17] C. Nédellec, C. Rouveirol, F. Bergadano, H. Adé, and B. Tausend. Declarative bias in ilp. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, Amsterdam, 1996.

[18] G. Silverstein and M. Pazzani. Learning relational clichés. In F. Bergadano, editor, *Proceedings of the IJCAI-93 Workshop on Inductive Logic Programming*, Chambery, France, 1993.

[19] G. Silverstein and M.J. Pazzani. Relational clichés: Constraining constructive induction during relational learning. In L.A. Birnbaum and G.C. Collins, editors, *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pages 203–207, San Mateo, CA, 1991. Morgan Kaufmann.

[20] A. Srinivasan, S. Muggleton, R.D. King, and M. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.

[21] B. Tausend. Representing biases for inductive logic programming. In F. Bergadano and L. De Raedt, editors, *Machine Learning: ECML-94*, Berlin Heidelberg New York, 1994. Springer.

A Definition of Syntax of Language

The suggested language $\mathcal{L}$ is a tuple $(\mathcal{MV}, \mathcal{MT}, \mathcal{MP}, \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{EL}, \mathcal{UL})$, where

- $\mathcal{MV}$ is a set of meta-variables,
- $\mathcal{MT}$ is a set of meta-types,
- $\mathcal{MP}$ is a set of meta-predicates,
- $\mathcal{V}$ is a set of variables,
- $\mathcal{T}$ is a set of types,

- $\mathcal{P}$ is a set of predicates,
- $\mathcal{EL}$ are expert-level declarations as defined below, and
- $\mathcal{UL}$ are user-level declarations as defined below.

In the following, $\mathcal{CC}$ is defined as $\{'0-1','0-n','1-1','0-n'\}$, the set of cardinality constraints, and $\mathcal{M}$ is defined as $\{'+','-','='\}$, the set of modes.

Expert-level declarations $\mathcal{EL}$ are a pair $(\mathcal{ML}, \mathcal{MS})$, where $\mathcal{ML}$ are declarations of the meta-literals, and $\mathcal{MS}$ are declarations of the meta-schemata.

$\mathcal{ML}$ are the meta-literals, i.e., expressions of the form:

$$\begin{aligned} &meta\_literal(mp(mv_1, ..., mv_n), \\ &\qquad [mv_1 : mt_1 : cc_1, ..., mv_n : mt_n : cc_n]), \end{aligned}$$

where $mp \in \mathcal{MP}$, $mv_i \in \mathcal{MV}$ and $mt_i \in \mathcal{MT}$, $cc_i \in \mathcal{CC}$.

$\mathcal{MS}$ are the meta-schemata, i.e., expressions of the form:

$$\begin{aligned} &meta\_schema((mp_1(mv_{1,1}, ..., mv_{1,m}), ..., mp_n(mv_{n,1}, ..., mv_{n,m})), \\ &\qquad [mv_{1,1} : mt_{1,1} : cc_{1,1} : mode_{1,1}, ..., \\ &\qquad mv_{n,m} : mt_{n,m} : cc_{n,m} : mode_{n,m}]), \end{aligned}$$

where $mp_i \in \mathcal{MP}$, $mv_{i,j} \in \mathcal{MV}$, $mt_{i,j} \in \mathcal{MT}$, $cc_{i,j} \in \mathcal{CC}$ and $mode_{i,j} \in \mathcal{M}$.

User-level declarations $\mathcal{UL}$ are a pair $(\mathcal{OD}, \mathcal{MD})$, where $\mathcal{OD}$ are object-level declarations, and $\mathcal{MD}$ are meta-level declarations.

$\mathcal{OD}$ are the object-level declarations, i.e., expressions of the form:

$$relation(p(v_1, ..., v_n), [v_1 : t_1, ..., v_n : t_n]),$$

where $p \in \mathcal{P}$, $v_i \in \mathcal{V}$ and $t_i \in \mathcal{T}$.

$\mathcal{MD}$ are the meta-level declarations, i.e., expressions of the form:

$$mp(p(v_1, ..., v_n), [v_1 : mt_1, ..., v_n : mt_n]),$$

where $mp \in \mathcal{MP}$, $p \in \mathcal{P}$, $v_i \in \mathcal{V}$ and $mt_i \in \mathcal{MT}$.

B Generate-And-Test Algorithm for Generating Schemata

The algorithm 1 proceeds in the following way: the meta-literals of the meta-schemata are processed one by one, from left to right, and one literal

```
$Conjunction \leftarrow true$                                                    (1)
$M \leftarrow initialize\_mapping(MS)$                                           (2)
for each meta-literal $ML$ in $MS$ do (in the order occurring in $MS$)           (3)
    $VarList \leftarrow []$                                                      (4)
    choose $P$ from the predicates matching the meta-predicate of $ML$           (5)
    $M' \leftarrow M$                                                            (6)
    for each argument position $N$ in the type declarations                      (7)
        of the relation corresponding to $P$, from left to right, do
        $MT \leftarrow$ meta-type at $N$ in meta-level declaration of the        (8)
                relation corresponding to $P$
        $T \leftarrow$  type at $N$ in declaration of the relation               (9)
                corresponding to $P$
        either:                                                                  (10)
            /* create new variables */
            choose $MV$ from mapping $M$, such that                              (11)
                $MV$ occurs in $ML$ $\wedge$                                     (12)
                $meta\_type(MV) = MT$ $\wedge$                                   (13)
                $MV$ is not yet marked as $set$ in $M$ $\wedge$                  (14)
                the upper bound of the cardinality constraints in $M'$           (15)
                  would not be violated if a new variable would be
                  assigned to $MV$
            "create" a new variable $NV$ and assign it to $MV$. In               (16)
                other words, add it, with its type $T$, to the entry
                of the meta-variable $MV$ in $M'$
            $VarList \leftarrow append(VarList, [NV])$                           (17)
        or:                                                                      (18)
            /* reuse variables */
            choose $EV$, a variable assigned to an $MV$ in $M$,                  (19)
            such that
                $MV$ occurs in $ML$ $\wedge$                                     (20)
                $meta\_type(MV) = MT$ $\wedge$                                   (21)
                $type(EV) = T$                                                   (22)
            $VarList \leftarrow append(VarList, [EV])$                           (23)
    end_for /* literal is fully variabilized */
    if (not all meta-variables of $ML$ are used properly for the                 (24)
        variabilization or any lower bound of a cardinality
        constraint in $M'$ is violated)
    then
        backtrack                                                                (25)
    $M \leftarrow M'$                                                            (26)
    mark all meta-variables of $ML$ as $set$ in $M$                              (27)
    $NewLiteral$ =.. $[P|VarList]$                                               (28)
    $Conjunction \leftarrow (Conjunction, NewLiteral)$                           (29)
end_for /* all meta-literals of meta-schema are processed */
$Schema \leftarrow create\_schema(Conjunction, M)$                               (30)
if $Schema$ has not yet been generated before                                    (31)
then
    $output(Schema)$                                                             (32)
if possible then backtrack                                                       (33)
```

Figure 1: Pseudocode of the procedure for generating schemata.

is created for each meta-literal. For each meta-literal, the basic task of the algorithm is the assignment of variables to meta-variables. This assignment is heavily constrained, e.g. by meta-types and by cardinality constraints. So, the algorithm has to perform some kind of constraint satisfaction.

The algorithm maintains a data structure M, containing the mapping or assignment of variables to meta-variables. This assignment can easily be illustrated by a "snapshot" of data structure M. Consider the following meta-schema:

```
meta_schema((arc_relation(A,B,C,D,E,F,G),
             node_relation(A,C,H,I,J)),
            [A:example:'1-1':'+',
             B:node:'1-1':'+',
             C:node:'1-1':'-',
             D:sym_arc_feature:'0-n':'-',
             E:sym_arc_feature:'1-1':'=',
             F:num_arc_feature:'0-n':'-',
             G:other_arc_arg:'0-n':'-',
             H:sym_node_feature:'0-n':'-',
             I:num_node_feature:'0-n':'-',
             J:other_node_arg:'0-n':'-']).
```

The following expression is a state of data structure M, the one after having processed the first meta-literal of the meta-schema. It contains one entry for each meta-variable of the meta-schema, each entry containing its meta-type, its cardinality constraints, its mode and the assigned variables:

```
[A:example:'1-1':'+':[T:chemical]:set,
 B:node:'1-1':'+':[U:atomid]:set:
 C:node:'1-1':'-':[V:atomid]:set,
 D:sym_arc_feature:'0-n':'-':[]:set,
 E:sym_arc_feature:'1-1':'=':[W:bondtype]:set,
 F:num_arc_feature:'0-n':'-':[]:set,
 G:other_arc_arg:'0-n':'-':[]:set,
 H:sym_node_feature:'0-n':'-':[]:not_set,
 I:num_node_feature:'0-n':'-':[]:not_set,
 J:other_node_arg:'0-n':'-':[]:not_set]
```

In the example, variable `T` of type `chemical` is assigned to meta-variable `A` of meta-type `example`. Note that the cardinality constraints are not violated in the example. The number of assigned variables is in accordance with the respective cardinality constraints. Our generate-and-test algorithm generates such (valid or invalid) data structures, and backtracks if, e.g., the cardinality constraints are not fulfilled.

Once a literal is generated, and the assignment of variables to the meta-variables is made properly, this assignment is fixed and cannot be changed afterwards. A meta-variable reused in another meta-literal means that *all* variables assigned to the meta-variable have to be reused in the corresponding literal as well. On the other hand, all newly introduced meta-variables have to be set properly, i.e. in accordance with the lower bounds of the cardinality constraints, with each meta-literal processed.

Since our goal was to keep the algorithm as simple as possible, we employ a generate-and-test algorithm for the assignment task. Usually several alternative assignments can be made during the construction of a schema. So, our algorithm makes heavy use of backtracking in the construction process. A smarter algorithm performing the assignment would probably make more informed choices and thus avoid some unnecessary backtracking.

The algorithm is non-deterministic, and all possible solutions are calculated. The number of solutions is finite, as the number of possible assignments of variables to meta-variables is finite.

C Proof Sketches

Proposition 1: *In the output of algorithm 1, the cardinality constraints are never violated.*

This holds if neither the lower bound (1.) nor the upper bound (2.) is ever violated.

1.) The lower bound of a cardinality constraint is never violated.

Condition (24) checks whether after an assignment the lower bound *is* violated. If it is, then (25) backtracks to a previous choice. Only if (24) is fulfilled, the algorithm proceeds to (26). So, at (26), this condition is never violated.

2.) The upper bound of a cardinality constraint is never violated.

There is just one statement (16) in the algorithm at all, where a variable is assigned to a meta-variable. If the condition is not violated before, the cardinality constraint will also not be violated if an inadmissible assignment does not take place at (16). Condition (15) guarantees, that such an assignment is never made at (16). So, after (16), the upper bound can never be violated, if it is not violated before. As the mapping is empty right from the beginning, the upper bound can never be violated by the algorithm.

Proposition 2: *Meta-types and types are always matched correctly.*

There are two statements where meta-level and object-level are matched: Statement (13) and statement (21). Both conditions have to be fulfilled, if an assignment takes place. So, trivially, the meta-types and types are always matched correctly.

Proposition 3: *Meta-types of both levels are always matched correctly.*

This just matters, when a meta-variable is reused in another literal. Reuse only occurs in the block after (18). As condition (22) has to be fulfilled before the assignment is made, proposition 3 also holds trivially, by algorithm design.

D Graph-Based Meta-Schemata

```
/* G R A P H - B A S E D   M E T A - S C H E M A T A */

/* EXPERT LEVEL */

/* meta-literals */

meta_literal(arc_relation(A,B,C,D,E,F),
                   [A:example:'1-1',
                    B:node:'1-1',
                    C:node:'1-1',
                    D:sym_arc_feature:'0-n',
                    E:num_arc_feature:'0-n',
                    F:other_arc_arg:'0-n']).

meta_literal(node_relation(A,B,C,D,E),
                   [A:example:'1-1',
                    B:node:'1-1',
                    C:sym_node_feature:'0-n',
                    D:num_node_feature:'0-n',
                    E:other_node_arg:'0-n']).

meta_literal(equal(A,B),
             [A:num_arc_feature:'1-1',
              B:num_arc_feature:'1-1']).

meta_literal(equal(A,B),
             [A:num_node_feature:'1-1',
              B:num_node_feature:'1-1']).

meta_literal(equal(A,B),
             [A:sym_arc_feature:'1-1',
              B:sym_arc_feature:'1-1']).

meta_literal(equal(A,B),
             [A:sym_node_feature:'1-1',
              B:sym_node_feature:'1-1']).



meta_literal(gteq(A,B),
             [A:num_arc_feature:'1-1',
              B:num_arc_feature:'1-1']).

meta_literal(gteq(A,B),
             [A:num_node_feature:'1-1',
              B:num_node_feature:'1-1']).
```

```
/* meta-schemata */

/*

  1) Introducing a "detached" new node and testing some property of it.

*/


meta_schema((node_relation(A,B,C,D,E,F)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'-',
                    C:sym_node_feature:'0-n':'-',
                    D:sym_node_feature:'1-1':'=',
                    E:num_node_feature:'0-n':'-',
                    F:other_node_arg:'0-n':'-']).

meta_schema((node_relation(A,B,C,D,E,F), gteq(E, G)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'-',
                    C:sym_node_feature:'0-n':'-',
                    D:num_node_feature:'0-n':'-',
                    E:num_node_feature:'1-1':'-',
                    F:other_node_arg:'0-n':'-',
                    G:num_node_feature:'1-1':'=']).

/*

  2) Testing the existence of an arc between two "existing" nodes.

*/

meta_schema((arc_relation(A,B,C,D,E,F)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'+',
                    C:node:'1-1':'+',
                    D:sym_arc_feature:'0-n':'-',
                    E:num_arc_feature:'0-n':'-',
                    F:other_arc_arg:'0-n':'-']).

/*

  3) Testing a property of an arc between two "existing" nodes.

*/
```

```
meta_schema((arc_relation(A,B,C,D,E,F,G)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'+',
                    C:node:'1-1':'+',
                    D:sym_arc_feature:'0-n':'-',
                    E:sym_arc_feature:'1-1':'=',
                    F:num_arc_feature:'0-n':'-',
                    G:other_arc_arg:'0-n':'-']).

meta_schema((arc_relation(A,B,C,D,E,F,G), gteq(E, H)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'+',
                    C:node:'1-1':'+',
                    D:sym_arc_feature:'0-n':'-',
                    E:num_arc_feature:'1-1':'-',
                    F:num_arc_feature:'0-n':'-',
                    G:other_arc_arg:'0-n':'-',
                    H:num_arc_feature:'1-1':'=']).

/*

  4) Testing the existence of an arc between a known node and
     some other yet unknown node.

*/

meta_schema((arc_relation(A,B,C,D,E,F),
             node_relation(A,C,G,H,I)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'+',
                    C:node:'1-1':'-',
                    D:sym_arc_feature:'0-n':'-',
                    E:num_arc_feature:'0-n':'-',
                    F:other_arc_arg:'0-n':'-',
                    G:sym_node_feature:'0-n':'-',
                    H:num_node_feature:'0-n':'-',
                    I:other_node_arg:'0-n':'-']).


/*

  5) Introducing a new node by means of an arc and test one
     of the arc's properties.

*/

meta_schema((arc_relation(A,B,C,D,E,F,G),
             node_relation(A,C,H,I,J)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'+',
                    C:node:'1-1':'-',
                    D:sym_arc_feature:'0-n':'-',
                    E:sym_arc_feature:'1-1':'=',
                    F:num_arc_feature:'0-n':'-',
                    G:other_arc_arg:'0-n':'-',
                    H:sym_node_feature:'0-n':'-',
                    I:num_node_feature:'0-n':'-',
                    J:other_node_arg:'0-n':'-']).

meta_schema((arc_relation(A,B,C,D,E,F,G),
             gteq(E, H),
             node_relation(A,C,I,J,K)),
                   [A:example:'1-1':'+',
                    B:node:'1-1':'+',
                    C:node:'1-1':'-',
                    D:sym_arc_feature:'0-n':'-',
                    E:num_arc_feature:'1-1':'-',
                    F:num_arc_feature:'0-n':'-',
                    G:other_arc_arg:'0-n':'-',
                    H:num_arc_feature:'1-1':'=',
                    I:sym_node_feature:'0-n':'-',
                    J:num_node_feature:'0-n':'-',
                    K:other_node_arg:'0-n':'-']).

/*

  6) Introducing a new node by means of an arc and test one
     of the new node's properties.

*/

meta_schema((arc_relation(A,B,C,D,E,F),
             node_relation(A,C,G,H,I,J)),
            [A:example:'1-1':'+',
             B:node:'1-1':'+',
             C:node:'1-1':'-',
             D:sym_arc_feature:'0-n':'-',
             E:num_arc_feature:'0-n':'-',
             F:other_arc_arg:'0-n':'-',
             G:sym_node_feature:'1-1':'=',
             H:sym_node_feature:'0-n':'-',
             I:num_node_feature:'0-n':'-',
             J:other_node_arg:'0-n':'-']).
```

```
meta_schema((arc_relation(A,B,C,D,E,F),
             node_relation(A,C,G,H,I,J),
             gteq(H,K)),
            [A:example:'1-1':'+',
             B:node:'1-1':'+',
             C:node:'1-1':'-',
             D:sym_arc_feature:'0-n':'-',
             E:num_arc_feature:'0-n':'-',
             F:other_arc_arg:'0-n':'-',
             G:sym_node_feature:'0-n':'-',
             H:num_node_feature:'1-1':'-',
             I:num_node_feature:'0-n':'-',
             J:other_node_arg:'0-n':'-',
             K:num_node_feature:'1-1':'=']).

/*

 7) Testing some "feature variable": equal to some constant?

*/

meta_schema(equal(A, B), [A:sym_node_feature:'1-1':'+',
                          B:sym_node_feature:'1-1':'=']).

meta_schema(gteq(A, B), [A:num_node_feature:'1-1':'+',
                         B:num_node_feature:'1-1':'=']).

meta_schema(equal(A, B), [A:sym_arc_feature:'1-1':'+',
                          B:sym_arc_feature:'1-1':'=']).


meta_schema(gteq(A, B), [A:num_arc_feature:'1-1':'+',
                         B:num_arc_feature:'1-1':'=']).

/*

 8) Testing the equality of two bound "feature variables".

*/

meta_schema(equal(A, B), [A:sym_node_feature:'1-1':'+',
                          B:sym_node_feature:'1-1':'+']).


meta_schema(gteq(A, B), [A:num_node_feature:'1-1':'+',
                         B:num_node_feature:'1-1':'+']).
```

```
meta_schema(equal(A, B), [A:sym_arc_feature:'1-1':'+',
                          B:sym_arc_feature:'1-1':'+']).


meta_schema(gteq(A, B), [A:num_arc_feature:'1-1':'+',
                         B:num_arc_feature:'1-1':'+']).



/* USER LEVEL */

/* object-level declarations */

relation(atm(A,B,C,D,E),
         [A:chemical,  B:atomid, C:element,
          D:atomtype, E:charge]).

relation(sym_bond(A,B,C,D),
         [A:chemical, B:atomid, C:atomid,
          D:bondtype]).

relation(equal(A,B), [A:element, B:element]).

relation(equal(A,B), [A:charge, B:charge]).

relation(equal(A,B), [A:atomtype, B:atomtype]).

relation(equal(A,B), [A:bondtype, B:bondtype]).

relation(gteq(A,B), [A:charge, B:charge]).


/* meta-level declarations */

node_relation(atm(A,B,C,D,E),
              [A:example,
               B:node,
               C:sym_node_feature,
               D:sym_node_feature,
               E:num_node_feature]).

arc_relation(sym_bond(A,B,C,D),
             [A:example, B:node, C:node, D:sym_arc_feature]).

equal(equal(A,B), [A:sym_node_feature, B:sym_node_feature]).
equal(equal(A,B), [A:sym_arc_feature,  B:sym_arc_feature]).

equal(equal(A,B), [A:num_node_feature, B:num_node_feature]).

equal(equal(A,B), [A:num_arc_feature, B:num_arc_feature]).

gteq(gteq(A,B), [A:num_node_feature, B:num_node_feature]).

gteq(gteq(A,B), [A:num_arc_feature, B:num_arc_feature]).

/* generated schemata */

schema(atm(A,B,C,D,E),
 [A:chemical:'+',B:atomid:'-',C:element:'-',
  D:atomtype:'=',E:charge:'-']).

schema(atm(A,B,C,D,E),
 [A:chemical:'+',B:atomid:'-',C:element:'=',
  D:atomtype:'-',E:charge:'-']).

schema((atm(A,B,C,D,E),gteq(E,F)),
 [A:chemical:'+',B:atomid:'-',C:element:'-',
  D:atomtype:'-',E:charge:'-',F:charge:'=']).

schema(sym_bond(A,B,C,D),
 [A:chemical:'+',B:atomid:'+',C:atomid:'+',D:bondtype:'-']).

schema(sym_bond(A,B,C,D),
 [A:chemical:'+',B:atomid:'+',C:atomid:'+',D:bondtype:'=']).

schema((sym_bond(A,B,C,D),atm(A,C,E,F,G)),
 [A:chemical:'+',B:atomid:'+',C:atomid:'-',D:bondtype:'-',
  E:element:'-',F:atomtype:'-',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,B,E,F,G)),
 [A:chemical:'+',B:atomid:'-',C:atomid:'+',D:bondtype:'-',
  E:element:'-',F:atomtype:'-',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,C,E,F,G)),
 [A:chemical:'+',B:atomid:'+',C:atomid:'-',D:bondtype:=,
  E:element:'-',F:atomtype:'-',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,B,E,F,G)),
 [A:chemical:'+',B:atomid:'-',C:atomid:'+',D:bondtype:'=',
  E:element:'-',F:atomtype:'-',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,C,E,F,G)),
 [A:chemical:'+',B:atomid:'+',C:atomid:'-',D:bondtype:'-',
  E:element:'=',F:atomtype:'-',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,C,E,F,G)),
 [A:chemical:'+',B:atomid:'+',C:atomid:'-',D:bondtype:'-',
  E:element:'-',F:atomtype:'=',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,B,E,F,G)),
 [A:chemical:'+',B:atomid:'-',C:atomid:'+',D:bondtype:'-'
  E:element:'=',F:atomtype:'-',G:charge:'-']).

schema((sym_bond(A,B,C,D),atm(A,B,E,F,G)),
 [A:chemical:'+',B:atomid:'-',C:atomid:'+',D:bondtype:'-',
  E:element:'-',F:atomtype:'=',G:charge:'-']).

schema(((sym_bond(A,B,C,D),atm(A,C,E,F,G)),gteq(G,H)),
 [A:chemical:'+',B:atomid:'+',C:atomid:'-',D:bondtype:'-',
  E:element:'-',F:atomtype:'-',G:charge:'-',H:charge:'=']).

schema(((sym_bond(A,B,C,D),atm(A,B,E,F,G)),gteq(G,H)),
 [A:chemical:'+',B:atomid:'-',C:atomid:'+',D:bondtype:'-',
  E:element:'-',F:atomtype:'-',G:charge:'-',H:charge:'=']).

schema(equal(A,B), [A:element:'+',B:element:'=']).
schema(equal(A,B), [A:element:'=',B:element:'+']).
schema(equal(A,B), [A:charge:'+',B:charge:'=']).
schema(equal(A,B), [A:charge:'=',B:charge:'+']).
schema(equal(A,B), [A:atomtype:'+',B:atomtype:'=']).
schema(equal(A,B), [A:atomtype:'=',B:atomtype:'+']).
schema(equal(A,B), [A:bondtype:'+',B:bondtype:'=']).
schema(equal(A,B), [A:bondtype:'=',B:bondtype:'+']).
schema(gteq(A,B), [A:charge:'+',B:charge:'=']).
schema(gteq(A,B), [A:charge:'=',B:charge:'+']).
schema(equal(A,B), [A:element:'+',B:element:'+']).
schema(equal(A,B), [A:charge:'+',B:charge:'+']).
schema(equal(A,B), [A:atomtype:'+',B:atomtype:'+']).
schema(equal(A,B), [A:bondtype:'+',B:bondtype:'+']).
schema(gteq(A,B), [A:charge:'+',B:charge:'+']).
```