MILES – a Modular Inductive Logic Programming Experimentation System

Irene Stahl *Birgit Tausend*

Institut für Informatik, Universität Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart

stahl@informatik.uni-stuttgart.de tausend@informatik.uni-stuttgart.de

January 10, 1994

Abstract

In this paper, we give a short technical description of the prototypical system MILES. MILES is a toolbox of ILP-methods and operators. It provides a flexible and powerful environment for handling, maintaining and inductively transforming a knowledge base of Horn clauses with respect to the examples. It is designed to support researchers in ILP in rapid prototyping of ILP methods and systems.

1 Introduction

The task of inductive logic programming (ILP) is to learn logic programs from classified examples in the presence of background knowledge. In recent years, a large variety of algorithms that tackle this problem have been developed and investigated. They mainly differ in the way the hypothesis space is searched, the operators for generalising or specialising hypotheses, the particular bias used to prune the search space, and the ability to recover from an overly strong bias.

In order to compare the different approaches and to experiment with known and new operators, control mechanisms, biases, heuristics and methods for shifting the bias, we developed MILES. MILES is a toolbox for experiments with ILP systems and methods. It offers a flexible representation for examples, background knowledge and hypotheses as well as procedures for handling and maintaining the knowledge base. It contains a large range of generalisation, specialisation and reformulation operators, procedures for initializing and evaluating the knowledge base, and a generic control for testing various combinations of operators.

MILES can be used for different goals. First, it allows to investigate the effects of the available operators on the knowledge base without specifying a specific control mechanism. Secondly, it facilitates the construction of specific ILP-systems by adding a control mechanism that uses some of the operators. A third class of experiments can be done by specifying a parametric ILP-system and studying the effects of different instantiations.

The generic control of MILES is a parametric ILP-system of that kind. It yields different specific ILP-methods according to its parameter instantiations. The operators and tools available in MILES allow to experiment with a large range of instantiations of the algorithm. They help to rapidly develop and test new approaches to ILP.

In the following, we give a short overview of the representation and operators of MILES. Then we describe the generic control, and give some example instantiations. At last, we present the X user interface of MILES, and conclude.

2 The Representation of MILES

The MILES system is an environment for inductive logic programming. Before we present the different inductive operators, we describe the knowledge base the operators work on.

The knowledge base of MILES consists of

- positive and negative examples,
- Horn clauses that constitute the background knowledge and hypotheses,
- information on argument types.

To realise closed-loop learning, we choose the same representation for the background knowledge and hypotheses. Examples and rules are stored as higher-order facts with a unique identifier. A depth-bounded iterative-deepening theorem prover allows to interpret the rules, and to deduce the examples from the theory.

2.1 Examples

An example is stored in the knowledge base as

$$ex(ID, Fact, Class)$$

where ID is the unique identifier. $Fact$ is a ground unit clause, and $Class$ either $+$ for positive or $-$ for negative examples.

MILES offers procedures for reading, storing and deleting examples. As only those procedures are visible to the user of the system, the actual implementation can be changed independently.

2.2 Horn Clauses

Both clauses of the background knowledge and hypothesis clauses are stored in the knowledge base as

$$known(ID, Head, Body, Clist, Label, evaluation(....))$$

where ID is the knowledge base key, and $Head$ and $Body$ are the head and the body of the clause. $Clist$ contains an alternative, but equivalent list representation of the clause that is advantageous for some operators. For example, for a clause with head H and body $B_1, .., B_n$ $Clist$ is $[H : p, B_1 : n, .., B_n : n]$. $Label$ allows to distinguish between background and hypothesis clauses. It may for example refer to the generating operator of the clause, or to the source file. The argument $evaluation$ contains information on the applicability of the clause in proving the examples.

Again, MILES offers a set of procedures for manipulating clauses. These procedures are the interface for the user, such that the actual implementation might be changed.

2.3 Argument Types

The specification of argument types for a predicate is optional and might be elicited automatically, if examples for the predicate are present. Argument types are stored as facts

$$type\_restriction(p(V_1, .., V_n), [type_1(V_1), .., typen(V_n)])$$

Each type $type_i$ is defined by clauses in the knowledge base of MILES. Besides the method for determining argument types mechanically, there are procedures for comparing types on generality, and for determining the type of a variable within a clause. These procedures can be employed during the generalisation or specialisation of hypotheses.

MILES contains procedures for initializing a knowledge base from files, saving and restoring the state of a knowledge base, and inspecting rules and examples.

3 The Operators of MILES

MILES contains five different kinds of operators.

- generalisation operators which generalise a single clause or sets of clauses
- specialisation operators which specialise a single clause or sets of clauses
- reformulation operators which perform an equivalent reformulation of the knowledge base
- preprocessing operators which initialize the knowledge base and hypotheses
- evaluation operators which measure the quality of the knowledge base

In the following, we give a short description of the implemented operators.

3.1 Generalisation Operators

Generalisation operators generalise single clauses or sets of clauses with respect to a generalisation model. Generalisation models formally define a generality relation between clauses and sets of clauses. Several generalisation models have been proposed in ILP, such as θ-subsumption [Plo70], generalised subsumption [Bun88], relative subsumption [Plo71, MF90] and logical implication. These generalisation models define the search space for the generalisation operators.

3.1.1 Least General Generalisations

Given a set of clauses, their least general generalisation (*lgg*) with respect to the generalisation model is of special interest for learning procedures. MILES contains *lgg*-operators for all generalisation models except for logical implication. However, in that case no unique or finite *lgg* needs to exist.

Three *lgg*-operators with respect to θ-subsumption have been implemented: the complete *lgg*-operator [Plo70] which involves the costly reduction under θ-subsumption, a weaker version *nr_lgg* that works without reduction and an operator *headed_lgg* which is specially suited for Horn clauses and fails if the input clauses have incompatible heads. An alternative generalisation operator with respect to θ-subsumption is the *gti*-operator that realizes generalisation through intersection [Jun92]. In contrast to *lgg*, the generalised clause is at most as long as the shortest input clause.

With respect to stronger generalisation models, two operators have been implemented. The first, *gen_msg* computes the least general generalisation with respect to generalised subsumption [Bun88]. As the background knowledge is involved, it generally results in complex clauses that have to be reduced in additional steps. The same holds for the second operator, *rlgg*, that determines the least general generalisation with respect to relative subsumption [Plo71, MF90]. It is realised by Muggleton und Feng's algorithm [MF90].

3.1.2 Inverse Resolution

Inverse resolution operators aim to invert single or multiple deductive resolution steps. According to the clauses given as input, $\mathcal{V}$- and $\mathcal{W}$-operators can be distinguished.

Given a parent clause B and a resolvent C, $\mathcal{V}$-operators determine the missing parent clause. Depending on whether the resolution literal in B is positive or negative, the missing parent clause is a generalisation with respect to generalised subsumption or realitive subsumption [Jun93]. The operators *absorption* [MB88] and *saturation* [Rou91] realised in MILES invert generalised subsumption, whereas *identification* [MB88] and *inverse_derivation* [Mug90] invert relative subsumption. The G_1-operator [Wir89] integrates both.

$\mathcal{W}$-operators start with a set of resolvent clauses $\{B_1, ..., B_n\}$ and construct a clause A and clauses $\{C_1, ..., C_n\}$ such that B_i results from resolving A with C_i on a fixed literal L in A. Since L is resolved away in B_i and nothing is known about its predicate symbol, a new predicate is invented. Three $\mathcal{W}$-operators are realised in MILES. The *intraconstruction*-operators of [MB88] and [Rou91] are a special case as the new predicate literal is assumed to be negative in A. The more general G_2-operator [Wir89] allows for both L being positive and negative in A.

3.1.3 Truncation

Truncation operators generalise clauses by dropping body literals. They realise a very simple generalisation with respect to θ-subsumption. Combined with saturation or inverse derivation, and inverse substitution, truncation operators allow for complete generalisations with respect to generalised or relative subsumption [Jun93]. That is, they allow to construct every clause that is more general than the input clause.

Heuristics and the knowledge about properties of logic programs control which body literals are actually removed. MILES contains five different truncation operators. The first, *truncate_facts*, is based on the observation that every body literal subsumed by a fact in the knowledge base is true regardless of the current proof. Therefore, the literal can be removed from the clause without changing the success set of the program.

The second operator, *truncate_unconnected*, starts from a similar observation described in [Rou91]. Unconnected literals, that is, literals that share no variables with the clause head or other connected literals, can be skipped without affecting the success set of the clause[1].

The third operator, *truncate_unconnecting*, is only slightly different. It drops a body literal if all other literals remain connected [Rou91]. However, this slight difference leads to varying results, as the following example will show.

Example 1: If a clause

$$p(X) \leftarrow q(X, V1), r(V2), s(V2, V1)$$

is given, then *truncate_unconnecting* removes the literal $r(V2)$, whereas *truncate_unconnected* does not change the clause.

The fourth truncation operator, *truncate_negation_based*, relies on the examples. It removes a body literal if the remaining clause still excludes all negative examples

[1] Except for the case that the clause has an empty success set.

[MF90]. Combined with an *lgg*-operation, negation-based truncation allows to remove redundant literals from the generalisation without sacrificing correctness with respect to the examples.

The last operator, *truncate_redundant*, leaves out all literals within a saturated clause that have been used for an elementary saturation step [Rou91]. This allows to implement absorption [MB88].

Example 2: Let

$$\begin{array}{ll} C: & bird(X) \leftarrow vulture(X), \\ D: & has\_beak(X) \leftarrow vulture(X). \end{array}$$

Then, saturation of D with respect to C yields

$$E: has\_beak(X) \leftarrow bird(X), \underline{vulture(X)}.$$

The *truncate_redundant*-operator removes $vulture(X)$ from E, resulting in the same clause as an absorption between D and C.

3.2 Specialisation Operators

Specialisation operators specialise a single clause with respect to a generalisation model, usually θ-subsumption. Four specialisation operators are realised in MILES. Three if them are part of Shapiro's *refinement operator* ρ_0 [Sha83]. Given a clause c, the specialisations of c that result from unifying variables within c, instantiating variables with terms and adding body literals are determined. Our implementation uses the argument types of the predicates involved, but it does not employ mode information, in contrast to Shapiro's operator.

The second specialisation operator implemented in MILES specialises a clause with a new predicate. It employs discrimination-based reduction [KNS92] to find the minimal set of argument variables for the new predicate.

3.3 Reformulation Operators

Reformulation operators perform equivalent transformations of the knowledge base that facilitate the learning task. The first operator realised in MILES is the *reduction* of clauses with respect to θ-subsumption [Plo70]. It helps to find the shortest equivalent form of a given clause. This is important especially for operators that construct very long and complex clauses as e.g. *rlgg* [MF90]. However, reduction with respect to θ-subsumption is quite costly. MILES contains different approximations, namely the truncation operators, that remove redundant body literals without performing a complete reduction.

The second reformulation operator implemented in MILES is *flattening* [Rou91]. It replaces n-ary functions with $(n+1)$-ary predicates throughout the knowledge base. The resulting program is function-free and allows for deciding logical implication and satisfiability. Furthermore, generalisation operators like saturation or absorption are more easily to apply as inverse substitutions are not necessary. The reverse *unflattening* operator restores the n-ary functors from the $(n+1)$-ary predicates.

3.4 Preprocessing Operators

Preprocessing operators are designed to elicit information given implicitly within the examples, and to initialize the hypothesis. The first operator determines the *argument types* for all example predicates. Argument types can be described by RUL-programs, and induced very efficiently from the given examples by the method described in [STW93]. The knowledge about types can be employed to prune the search space during generalisation or specialisation.

The second preprocessing operator implemented in MILES determines a set of *clause heads* that cover all positive examples in the knowledge base by the method described in [STW93]. These clause heads give important structural information about the disjunctive cases of the predicates. They can be used as initial hypothesis that is specialised further.

3.5 Evaluation of the Knowledge Base

Evaluation operators measure the quality of the knowledge base according to different criteria. A very simple measures realised in MILES is the *syntactic size* of the knowledge base.

A more complex operation evaluates the current theory with respect to the given examples. It determines the *evaluation* of the clauses in the knowledge base, i.e. the positive and negative examples they cover and the proofs of examples they have been used in. This information stored for each clause can be used to compute commonly employed measures as accuracy or information gain of the current theory with respect to the examples.

Additionally, MILES contains predicates that check whether the current theory is *complete* or *consistent* with respect to the examples. If neither is the case, procedures that detect the culprit are implemented in MILES. The first one, fp [Sha83], implements an oracle-free version of Shapiro's contradiction-backtracing algorithm that localises overgeneral clauses within a theory. These clauses need to be specialised to make the theory consistent. The second one, ip [Sha83], implements Shapiro's algorithm for diagnosing finite failures on the examples. It returns a set of ground facts that should be covered in order to make the theory complete. Both procedures can be used to provide relevant input to the generalisation and specialisation operators of MILES.

4 Generic Control Mechanism

To test the effects of different operator combinations, MILES contains a generic control procedure that can be instantiated to actual ILP algorithms. It is basically the same as the generic concept learning algorithm GENCOL [Rae92], except that it offers the possibility to invent new predicates. Because of its vicinity to GENCOL, it is called GENCON for GENeric CONtrol. Figure 1 shows the GENCON algorithm. Twelve parameter procedures, underlined in figure 1, control the hypothesis space, the search paradigm and the decision criteria for predicate invention of GENCON.

The central idea of GENCON is to mark the hypotheses in *partial_sols* as *active* and *passive*. *Active* programs are subject to further generalisation or specialisation steps, whereas *passive* ones have already been tried. Programs marked as *passive* are potential starting points for predicate invention.

The parameter procdure *initialize* in figure 1 initializes the set of partial solutions *partial_sols* to $\{(PS_1, Mark_1), ..., (PS_n, Mark_n)\}$. It should be based on the examples present in the knowledge base.

The procedures *Stop_Criterion* and *Quality_Criterion* express the success criterion of the algorithm. The procedure *Stop_Criterion* decides whether the set *complete_sols* contains a satisfactory solution. In that case, the algorithm stops and outputs one or all of the complete solutions depending on the implementation of *output*. The procedure *Quality_Criterion* checks whether the current hypothesis PS is satisfactory and should be added to *complete_sols*. In that case, it might

```
Given: B, E⊕, E⊖
Algorithm:
partial_sols := initialize(E⊕, E⊖, B)
all PS ∈ partial_sols marked active
complete_sols := φ
while not(Stop_Criterion(complete_sols)) do
    PS := select(partial_sols)
    if Quality_Criterion(PS)
    then complete_sols := complete_sols ∪ {PS}
         partial_sols := update(partial_sols)
    else if active(PS)
         then one_of( → partial_sols := add(partial_sols, spec(PS))
                      → partial_sols := add(partial_sols, gen(PS)))
                all PS' ∈ spec(PS)(gen(PS)) marked active
         else partial_sols := add(partial_sols, learn_newp(PS))
    mark PS as passive
    partial_sols := filter(partial_sols)
Output: output(complete_sols)
```

Figure 1: Generic Control of MILES: GENCON

be necessary to update *partial_sols* with the procedure *update*. For example, to implement a greedy covering strategy, both *partial_sols* and *complete_sols* contain clauses as elements, the latter consistent, the first possibly overgeneral ones. The *Stop_Criterion* is fullfilled as soon as the clauses in *complete_sols* cover all positive examples, whereas the *Quality_Criterion* is true if the clause PS is consistent with respect to the negative examples. In that case, *partial_sols* needs to be reset to containing only one active clause head to be specialised.

The procedures *select*, *add* and *filter* constitute the search paradigm. The procedure *select* selects one of the current hypotheses to be generalised or specialised, *add* adds the newly generated hypotheses to the set of partial solutions, and *filter* filters the most promising ones among them. For example, to implement breadth-first search, *select* always chooses the first *active* element of *partial_sols*, *add* appends the new partial programs to *partial_sols*, and *filter* is the identity.

Additionally, *select* controls when predicate invention is performed. If it chooses active programs as long as any are present, and passive ones only after all programs in *partial_sols* have become marked passive, it implements finite axiomatisability. This is because as long as active programs are present the hypothesis space is not yet completely explored. As soon as all programs in *partial_sols* are marked passive, the hypothesis space is exhausted and needs to be enlarged. If it selects passive programs though active ones are present, it implements a heuristic criterion for predicate invention. Active programs indicate further options for generalisation or specialisation that are discarded in sake of a more promising new predicate.

The predicate *one_of* decides whether the current hypothesis should be generalised or specialised. A common implementation as e.g. in MIS [Sha83] generalises if PS is incomplete, and specialises if PS is inconsistent. The generalisation and specialisation operators *gen* and *spec* determine the generalisations and specialisations of PS with respect to the bias.

The procedure *learn_newp* corrects the overgeneral and/or overspecific partial solution PS with new predicates. For finding the definition of the new predicates, a recursive call of the whole induction algorithm might be necessary.

GENCON provides a simple and very general control, but leaves most work to the implementation of the parameter predicates. However, the large amount of operators and heuristics in MILES that can be used to instantiate the parameters reduces this implementational work a lot.

5 Simple Instantiations of the Generic Control

Currently, three instantiations of GENCON are included in the MILES toolbox, one for RUL-programs, one for constrained programs, and a FOIL-like algorithm. They demonstrate the interaction between the different parameters of GENCON.

5.1 RUL-Programs

Regular unary logic (RUL) programs [YS91] are logic programs that allow to describe syntactic argument types. They contain only unary predicates and allow for non-variable argument terms only in the clause heads. The head arguments of clauses of the same predicate must differ in their function symbol. Additionally, every variable in a clause must occur exactly once in the head and once in the body.

The extensions of predicates defined by RUL-programs are regular sets. Therefore, they can be efficiently compared on generality. Regular unary predicates can be sorted in a complete lattice with respect to set union and intersection of their extensions. Methods for comparing regular predicates are included in the argument type module of MILES.

To obtain an instantiation of GENCON for inducing RUL-programs, the parameters are set as follows:

- $initialize(E^{\oplus}, \phi, B) = \{(heads(E^{\oplus}), active)\}$ where the heads for $E^{\oplus}$ are defined as $\{lgg^2(\{p(f(...)) \in E^{\oplus}\}) \mid p \text{ predicate, } f \text{ functor in } E^{\oplus}\}$.

  RUL-programs for argument types are induced from positive examples only. Therefore, the second argument of *initialize* is the empty set.

- $Stop\_Criterion(complete\_sols) \equiv (|complete\_sols| = 1)$

- $select(partial\_sols)$ = the most specific active $PS \in partial\_sols$, if any. Else the most specific passive $PS \in partial\_sols$.

- $Quality\_Criterion(PS) \equiv (B \cup PS \vdash E^{\oplus})$ and PS is a RUL-program

- $update(partial\_sols) = partial\_sols$

- $one\_of \equiv$ if $B \cup PS \vdash E^{\oplus}$ then specialise, else generalise

- $add(partial\_sols, PSL) = partial\_sols \cup PSL$

- $spec(PS)$: localise a clause $C \in PS$ where $\exists\ V \in vars(head(C)) - vars(body(C))$ and specialise C by adding a body literal $p(V)$ that is true for all example instantiations of the clause

- $gen(PS)$: localise a clause $C \in PS$ where

  $$\{e \in E^{\oplus} \mid \exists \sigma head(C)\sigma = e\} - \{e \in E^{\oplus} \mid B \cup PS \vdash e\} \neq \phi,$$

  and remove its body literals.

[2] [Plo70]

- $learn\_newp(PS)$: for each clause $C \in PS$ with $V_C = vars(head(C)) - vars(body(C)) \neq \phi$, and for each $V \in V_C$, add a literal $newp_V(V)$ to the body of C. The instantiations $E^{\oplus}_V = \{newp_V(V)\sigma \mid \exists e \in E^{\oplus} head(C)\sigma = e\}$ are added to $E^{\oplus}$, and $heads(E^{\oplus}_V)$ to PS.

- $filter(partial\_sols) = partial\_sols$

- $output(complete\_sols)$: remove and replace all equivalent new predicates, then add the resulting clauses to the knowledge base of MILES. Removing equivalently defined new predicates is necessary to improve the readability of the program. The equivalence of new predicates can be checked with the methods for comparing regular unary predicates.

5.2 Constrained Programs

The restriction to constrained clauses, that is clauses without existential variables, is widely used in ILP and deductive data bases. In particular for ILP-systems this restriction is very useful as it avoids the combinatorial problems with existential variables.

The following parameter instantiation for GENCON leads to a method for inducing constrained programs:

- $initialize(E^{\oplus}, E^{\ominus}, B) = \{(clause\_heads(E^{\oplus}), active)\}$ where the clause heads for the predicates in $E^{\oplus}$ are determined as described in [STW93].

- $Stop\_Criterion(complete\_sols) \equiv (|complete\_sols| = 1)$

- $select(partial\_sols) =$ the first active $PS \in partial\_sols$, if any. Otherwise the passive $PS \in partial\_sols$ that covers the fewest negative examples.

- $Quality\_Criterion(PS) \equiv ((B \cup PS \vdash E^{\oplus}) \wedge (B \cup PS \not\vdash E^{\ominus}))$

- $update(partial\_sols) = partial\_sols$

- $one\_of \equiv$ if $B \cup PS \vdash E^{\oplus}$ then specialise, else generalise

- $add(partial\_sols, PSL) = partial\_sols \cup PSL$

- $spec(PS)$: localise an overgeneral clause $C \in PS$ with fp, and specialise C by adding a constrained body literal.

- $gen(PS)$: localise a clause $C \in PS$ where

$$\{e \in E^{\oplus} \mid \exists\sigma head(C)\sigma = e\} - \{e \in E^{\oplus} \mid B \cup PS \vdash e\} \neq \phi,$$

  and greedily remove its body literals until the missing examples are covered.

- $learn\_newp(PS)$: localise an overgeneral clause $C \in PS$ with fp and specialise it with a constrained new predicate. Abductively add the instantiations of the new predicate to the examples, and recursively invoke the algorithm on them.

- $filter(partial\_sols) = partial\_sols$

- $output(complete\_sols) =$ add the clauses in *complete_sols* to the knowledge base of MILES.

5.3 FOIL

FOIL [Qui90] learns function-free logic programs with a greedy-covering strategy guided by the information gain heuristic. To obtain a FOIL-like algorithm as instantiation of GENCON, the parameters are set as follows:

- $initialize(E^{\oplus}, E^{\ominus}, B) = \{(most\_general\_term(E^{\oplus}), active)\}$ where the most general term for $E^{\oplus}$ is $p(X_1, .., X_n)$ for the target predicate p/n. $E^{\oplus}$ must not contain examples for different predicates.
- $Stop\_Criterion(complete\_sols) \equiv (B \cup complete\_sols \vdash E^{\oplus})$, or the encoding length of $complete\_sols$ exceeds that of the examples.
- $select(partial\_sols)$ = the best active $PS \in partial\_sols$ according to the information gain heuristic, if any. Fails else.
- $Quality\_Criterion(PS) \equiv (B \cup PS \not\vdash E^{\ominus})$
- $update(partial\_sols) = \{(most\_general\_term(E_r^{\oplus}), active)\}$ for the positive examples not yet covered by $complete\_sols$.
- one_of : always specialise
- $add(partial\_sols, PSL) = partial\_sols \cup PSL$
- $spec(PS)$: specialise PS by adding a body literal
- $gen(PS)$: —
- $learn\_newp(PS)$: —
- $filter(partial\_sols)$: remove all clauses from $partial\_sols$ violating the encoding length restriction
- $output(complete\_sols)$ = add the clauses in $complete\_sols$ to the knowledge base of MILES.

6 User Interface

The X user interface of MILES, X-MILES, is built on top of the ProXT interface of QUINTUS prolog v3.1.1. ProXT is a prolog interface to the Motif widget set and the X Toolkit (Xt). Widgets are ready-made graphical components for building user interfaces, for example menus, dialog boxes, scroll bars and command buttons. The X Toolkit provides routines for creating and using such widgets. ProXT provides access from prolog to all the widgets in Motif and the Xt routines necessary for using them [Qui91].

Figure 2 shows the complete X interface of MILES. It is separated in six different parts. The *knowledge base area* presents the examples and the current set of rules. The *editor area* allows to edit, add or change examples and rules. The *command area* provides direct access to prolog, whereas the *message area* displays system messages about the state and the result of the computation. The *function area* contains groups of learning operators that might be applied to the current knowledge base. The *argument area* provides interactive facilities to specify the arguments of a learning operator. In the following, the functionality of the separate parts is discussed in detail.

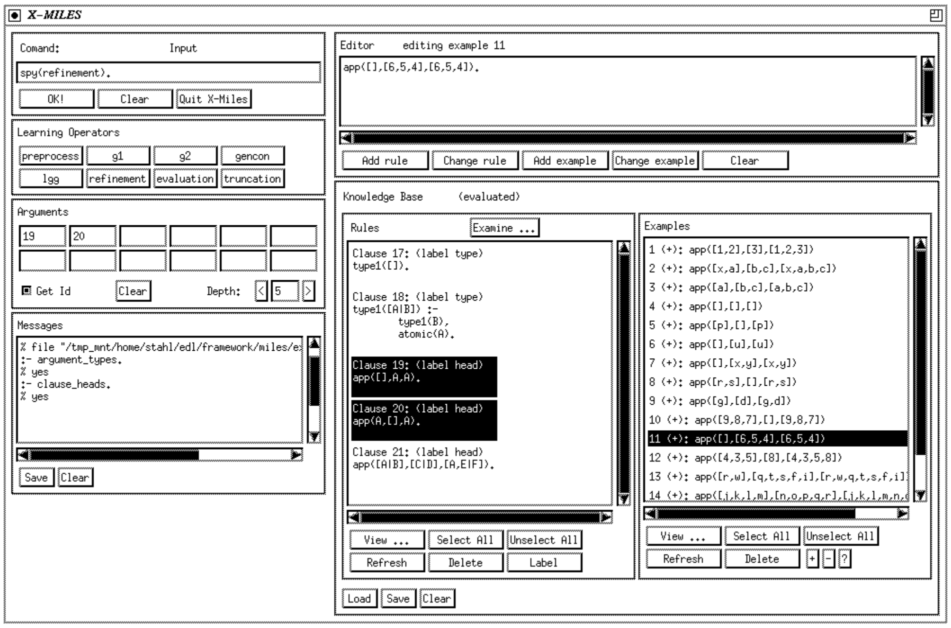

Figure 2: X-MILES

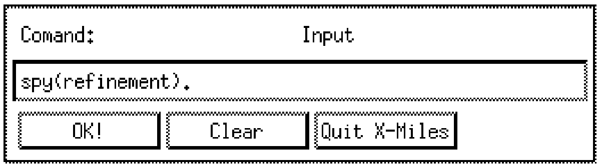

Figure 3: The Command Area

6.1 Prolog Interaction in the Command Area

The command area provides direct access to prolog. It allows for example to directly invoke operators not yet included in the interface, to turn on the debugger or to set spy points. These capabilities are an important precondition for the easy extendability of MILES and X-MILES.

Figure 3 shows the command area in more detail. The central part is a single-line editable window that allows to type in prolog commands as if directly at the prolog prompt. Hitting the return key or pressing the *OK!* button invokes the command. The status of the command line is indicated above. *Input* means that the command line allows input, *Yes* that a command has been successfully performed, *No* that it has failed, and *Exit* that it has been interrupted. The *Clear* button below the command line empties the command input, whereas the *Quit X-Miles* button returns to the prolog prompt.

6.2 Knowledge Base Area

The knowledge base of MILES consists basically of rules and examples. Accordingly, the knowledge base area of X-MILES shown in figure 4 contains an area for rules and for examples. The status of the knowledge base is indicated next to the knowledge base label. The status *evaluated* means that the rules are evaluated with respect to the examples, i.e. their applicability and coverage has been determined, whereas

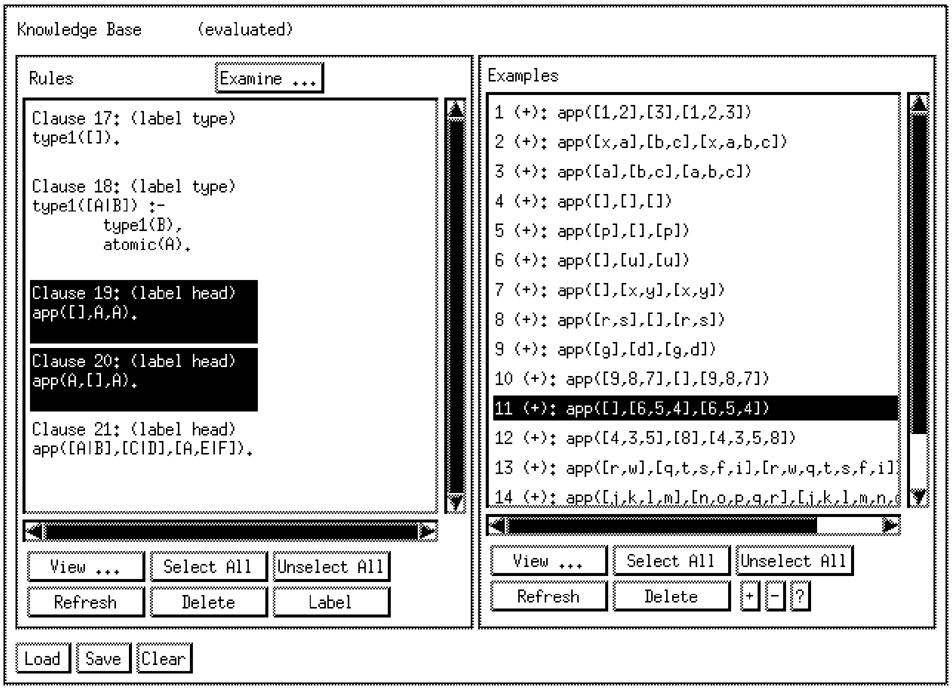

Figure 4: The Knowledge Base Area

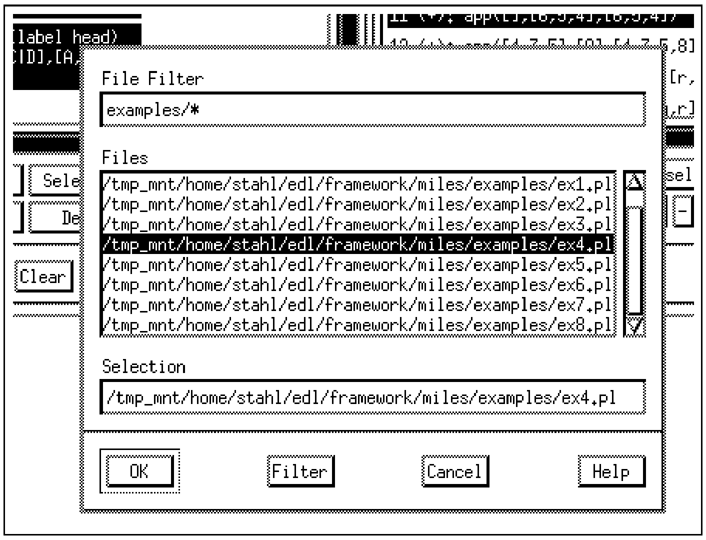

Figure 5: A File Selection Dialog

not evaluated indicates that the knowledge base has been changed since the last evaluation.

The utilities that apply to both examples and rules are realised as buttons in the bottom row of the knowledge base area. *Load* and *Save* open a file selection dialog as e.g. in figure 5. The files to be loaded may be either `.qof` files resulting from saving a knowledge base, or prolog files containing clauses, examples and type restrictions. *Save* saves the current knowledge base in `.qof` format. The *Clear* button empties the knowledge base of MILES.

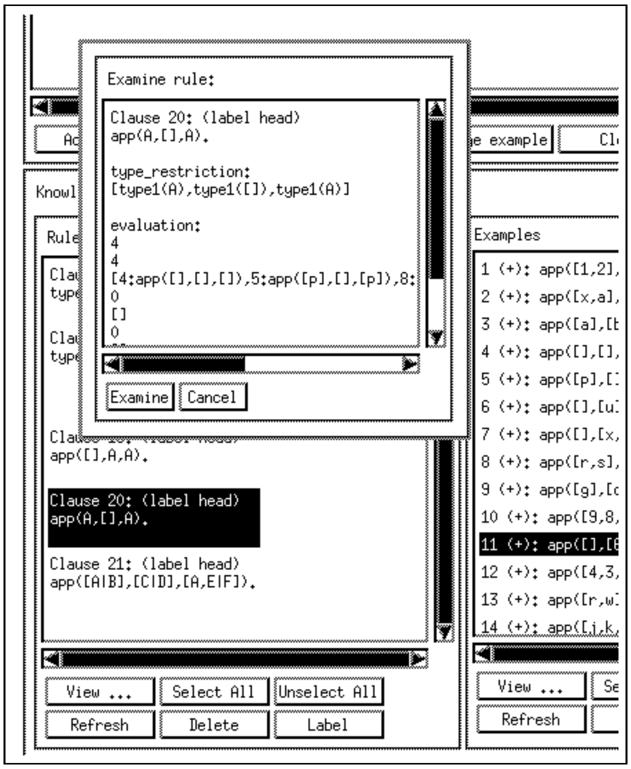

Figure 6: Examining a Rule

6.3 Rule Area

The rule area displays the unique identifier, the label, and head and body for each clause in the knowledge base. Further information on rules can be obtained by selecting a rule and pressing the *Examine* button in the title row. Figure 6 shows the resulting window. It displays the type restriction of the clause head, if any, and the evaluation of the clause.

Rules can be selected simply by a mouse click, or by the *Select All* and *Unselect All* buttons in the button row below. The *Delete* button deletes all selected rules, whereas *Refresh* updates the presentation of the rules according to the knowledge base of MILES. This might be necessary if a command does not activate the refresh callback.

The *Label* button provides a facility to change the label of the rules selected. For example, figure 7 shows how to change the label of two rules resulting from the clause-heads operator to `usr`.

The *View* button allows to specify a subset of the rules to be displayed. This is advantageous in case of large knowledge bases where creating and maintaining all rule widgets might become costly. The view-rules dialog in figure 8 displays all existing labels and head predicates. They can be transfered by mouse clicks to the viewed labels and head predicates area. *Min* and *Max* are the lower and upper bounds for clause identificators to be displayed. The *View* button activates the restrictions expressed in the viewed-part and by *Min* and *Max*, whereas *View All* deactivates them and displays all rules.

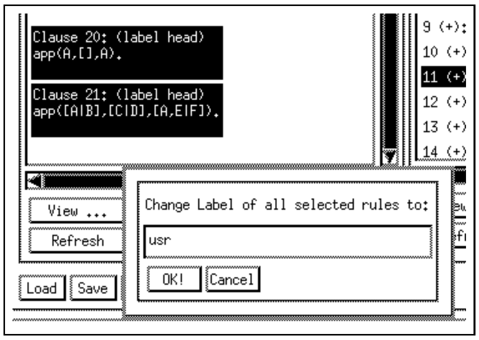

Figure 7: Changing Rule Labels

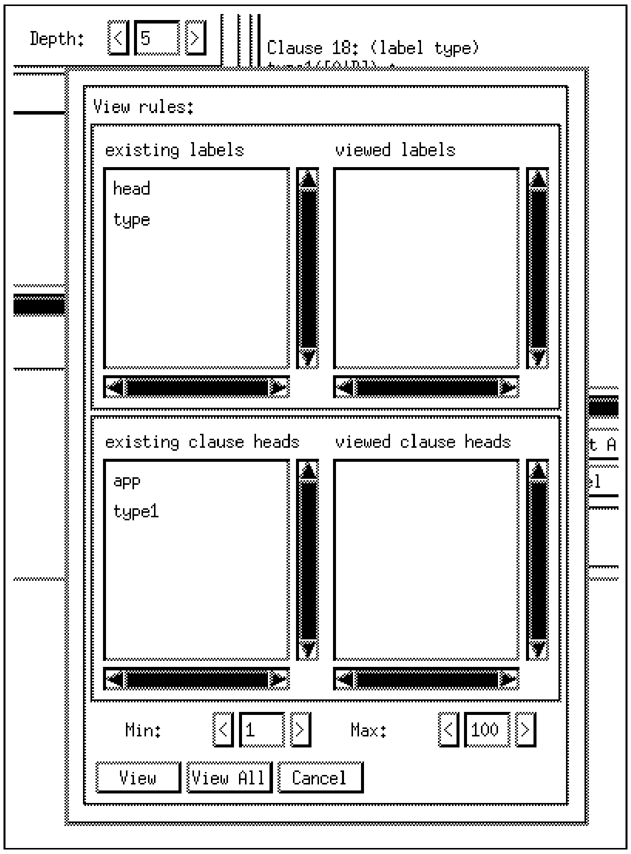

Figure 8: The View Rule Dialog

6.4 Example Area

The design of the example area is almost identical to that of the rule area. For each example, its unique identifier, its classification and the fact itsself is displayed. *Select All*, *Unselect All*, *Refresh* and *Delete* work exactly as for rules. Instead of changing the labels of a rule, the classification of the selected examples can be changed by pressing the *+*, *-* or *?* button. The view-example dialog in figure 9 is similar to that for rules, except that only the existing predicates are displayed. This is because examples are not labeled.

The knowledge base area is basically passive. It displays the contents of the knowledge base, but allows only minor changes on them. For really changing rules and

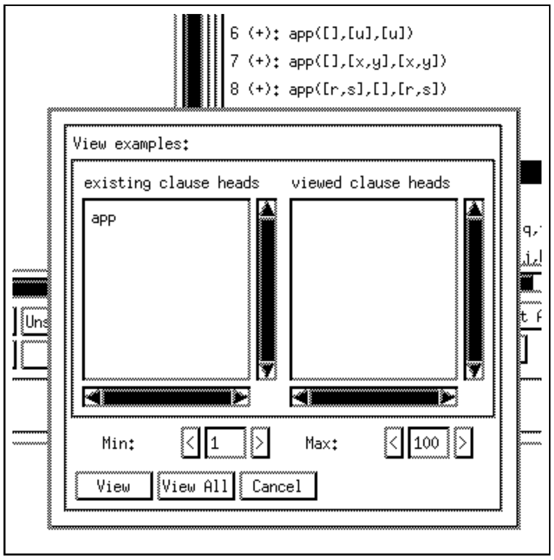

Figure 9: The View Example Dialog

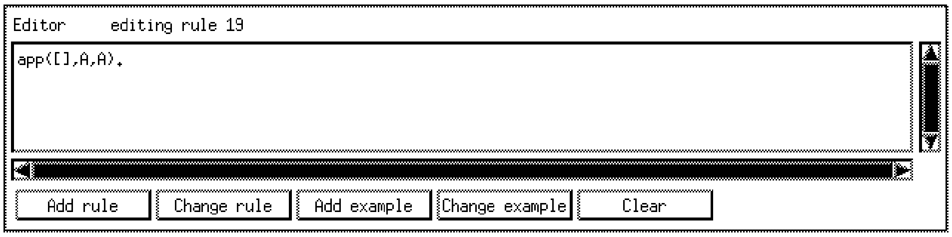

Figure 10: The Knowledge Base Editor

examples, an editor is necessary.

6.5 Knowledge Base Editor

The knowledge base editor shown in figure 10 basically consists of a multi-line editable and scrollable area. The buttons below allow to add new rules and examples that have been typed in. *Add Rule* and *Add Example* perform a simple syntax check and create a new unique identifier under which the rule, respectively the example, is stored. The label of the new rule defaults to `usr`, the classification of the new example to `'?'`.

Besides simply typing in new rules and examples, existing rules and examples can be copied into the editor area by double clicking them in the knowledge base area. After modification, either they can be added as new rules or examples, or they are used to overwrite the old definition. The latter is done with the *Change Rule* and *Change Example* buttons.

6.6 Function and Argument Area

The function area provides access to the different learning operators of MILES that might be applied to the current knowledge base. Because the large range of available operators cannot be displayed all at once, they are grouped together in menus that pop up on pressing the according push button, as eg. in figure 11.

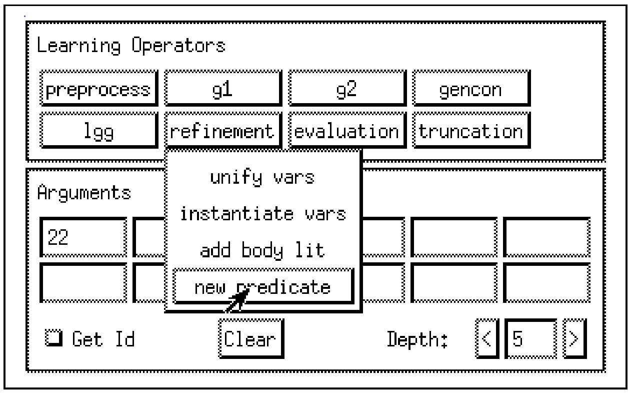

Figure 11: The Function Area

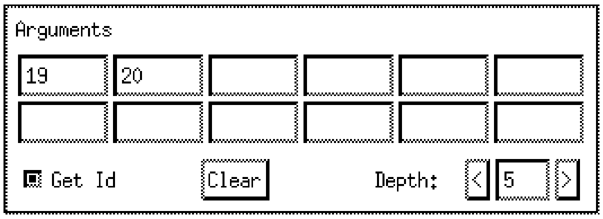

Figure 12: The Argument Area

The arguments of an operator are specified in the argument area shown in figure 12. A special argument is *Depth* that allows to specify globally the depth bound for operators like *saturation* or *rlgg*. Each of the twelve other argument positions might be either edited directly, or filled by first activating the toggle button *Get Id*, and then clicking the rules or examples in the knowledge base area on which the operator shall be applied. Of course, for specifying the arguments properly, one must know the number and type of them for each operator.

These can be found in the configuration file **xmiles_functions.pl**. This file allows to simply change or extend the function area of X-MILES. It contains easy-to-read and easy-to-add descriptions of the available operator groups and operators.

An operator group is defined by the expression

```
group( <group-name>, [ <operator-name>, <operator-name>, .... ] ).
```

This creates a push button labeled by **<group-name>** which pops up a menu of the operator names on being pressed.

A single operator is defined by

```
operator(  <operator-name>, <predicate-name>, InOutPattern,
           InTypeChecks, OutPredicates, KBChanges  )
```

The operator name must be one of the operator names in a group definition. The predicate name is the name of the procedure in MILES that is called on selecting the operator from the menu. **InOutPattern** specifies the input- and output arguments the predicate is called with. It is a list containing some of the following terms:

- **xmArg1 - xmArg12** for input arguments. They are set to the values of the according widgets in the argument area.
- **xmDepth** for the depth bound of the operator. It is set to the value of *Depth* in the argument area.

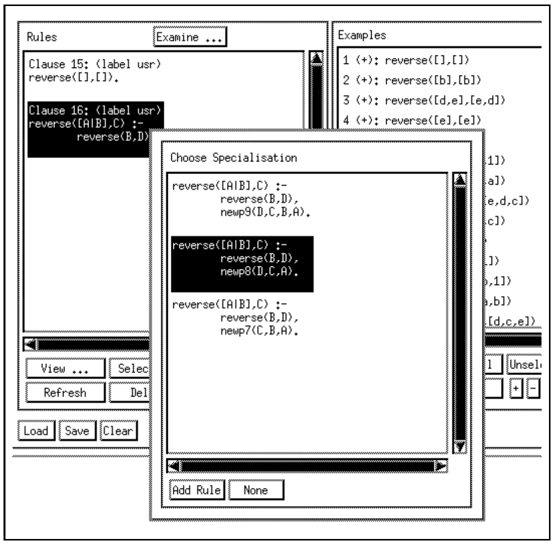

Figure 13: The Specialisation Dialog

- `xmOut` for output arguments.

The input arguments can be checked on their type. `InTypeChecks` is a list of unary predicates for each input argument, e.g. `isRule` or `isExample`.

The output argument is passed on to each of the predicates specified in `OutPredicates`. These might for example display several alternative solutions, or write informative text to the message area.

Refreshing the knowledge base area is controlled by `KBChanges`. This is either `[]`, `[rules]` if the rule area needs to be refreshed, `[examples]` if the example area needs a refresh, or `[rules, examples]` for both. This helps to avoid superfluous refreshs.

As an example, in figure 11 *New Predicate* is one of the operators of the refinement group. It is defined as follows:

```
operator( 'New Predicate', spec_with_newpred, [xmArg1,xmOut],
          [isRule], resultAddNewpred, [rules] ).
```

The procedure `resultAddNewpred` pops up the window in figure 13 that displays all potential specialisations with new predicates. The user might either select one and add it to the knowledge base through the *Add Rule* button, or reject them all by the *None* button.

The configuration file provides large a part of the flexibility and extendability of X-MILES. It allows to add or change operators without much knowledge about the realisation of the interface.

6.7 System Messages

The message area displays system messages in a scrollable window. It serves as a kind of protocol, as every prolog goal invoked through X-MILES is reported. Additional information, for example *Yes* for success, is prefixed with %. This allows to consult the message file in order to restore the knowledge base at a certain stage of interaction.

To save the message file, the *Save* botton below the message area in figure 14 must

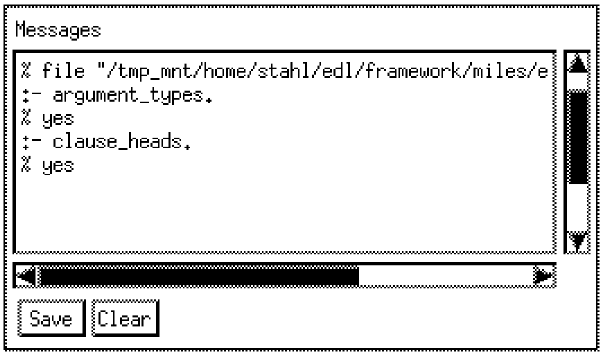

Figure 14: The Message Area

be used. It creates a file `xmProtokol.sav`. The *Clear* button empties the message window. This means the begin of a new logical X-MILES session.

7 Conclusions

MILES is a prototypical test-system for ILP. It offers a flexible and powerful environment of operators for handling, maintaining and inductively transforming a knowledge base of Horn clauses with respect to the examples. It is designed to support researchers in ILP in rapid prototyping of ILP-methods and -systems.

However, MILES itself is not an ILP-system in the sense that the user can simply input examples and get back a theory. Only instantiations of ILP-algorithms generated on the top of MILES might show that behaviour. Thus, MILES is more appropriate for people working on ILP-methods itself than for people applying them in practice.

Future work will concentrate on predicate invention techniques and the integration of a declarative representation language for biases that allows to express biased generalisation and specialisation operators for GENCON more easily.

Acknowledgements

This work has been supported by the European Community ESPRIT project ILP (Inductive Logic Programming).We want to thank Bernhard Jung, Markus Müller and Thorsten Volz for their implementational work on MILES and X-MILES.

References

[Bun88] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36:149–176, 1988.

[Jun92] B. Jung. Generalisierungsoperatoren bei der induktiven logischen Programmierung. Diplomarbeit nr. 940, Fakultät Informatik, Universität Stuttgart, 1992.

[Jun93] B. Jung. On inverting generality relations. In *Proc. of the Third International Workshop on Inductive Logic Programming ILP-93*, Technical Report, IJS-DP-6707. J. Stefan Institute, 1993.

[KNS92] B. Kijsirikul, M. Numao, and M. Shimura. Discrimination-based constructive induction of logic programs. In *Proc. of the 10th National Conference on AI*, 1992.

[MB88] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Fifth International Conference on Machine Learning*. Morgan Kaufmann, 1988.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.

[Mug90] S. Muggleton. Inductive logic programming. In *First Conference on Algorithmic Learning Theory*, Tokio, October 1990. Ohmsha.

[Plo70] G. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, Edinburgh, 1970.

[Plo71] G. Plotkin. A further note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 6, pages 101–124. Edinburgh Univeristy Press, Edinburgh, 1971.

[Qui90] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Qui91] Quintus Corporation, Palo Alto, California. *Quintus Prolog X Window Interface*, February 1991.

[Rae92] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach*. Academic Press, 1992.

[Rou91] C. Rouveirol. *ITOU: Induction de Théories en Ordre Un*. PhD thesis, Université Paris Sud, Centre d'Orsay, 1991.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[STW93] I. Stahl, B. Tausend, and R. Wirth. Two methods for improving inductive logic programming systems. In *Machine Learning: ECML-93, European Conference on Machine Learning, Wien, Austria*. Springer, 1993.

[Wir89] R. Wirth. Completing logic programs by inverse resolution. In *Fourth European Working Session on Learning*. Pitman, 1989.

[YS91] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, (10):125–153, 1991.