Learning in Logic with RichProlog *

Eric Martin[1], Phuong Nguyen[1], Arun Sharma[1], and Frank Stephan[2]

[1] School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia,
{emartin, ntp, arun}@cse.unsw.edu.au
[2] Universität Heidelberg, 69121 Heidelberg, Germany,
fstephan@math.uni-heidelberg.de

Abstract. Deduction and induction are unified on the basis of a generalized notion of logical consequence, having classical first-order logic as a particular case. RichProlog is a natural extension of Prolog rooted in this generalized logic, in the same way as Prolog is rooted in classical logic. Prolog can answer Σ_1 queries as a side effect of a deductive inference. RichProlog can answer Σ_1 queries, Π_1 queries (as a side effect of an inductive inference), and Σ_2 queries (as a side effect of an inductive inference followed by a deductive inference). RichProlog can be used to learn: a learning problem is expressed as a usual logic program, supplemented with data, and solved by asking a Σ_2 query. The output is correct in the limit, *i.e.*, when sufficient data have been provided.

1 Introduction

Enriching the expressive power of the logical language that subsumes the theory of Logic Programming has long been an active area of research. Examples of such approaches include consideration of negative literals in the body of clauses, interpreted either as negation as failure or as 'true' negation (for surveys see [3, 12]), and answering queries more complex than existentially quantified conjunctions of atoms [7]. All these investigations, however, have been in the realm of classical deductive logic. Since the class of Prolog programs determines an acceptable indexing of the class of all partial recursive functions, it might be argued that the fragment of classical logic that deals with rules (definite Horn clauses) and queries is expressive enough for all purposes. This is certainly true for the task of computing; but not for the task of computing by writing programs in a declarative manner — the *raison d'être* of Prolog.

To illustrate the above point, let us consider the task of implementing a learning strategy. Learning is not deducing. Indeed, deductive inferences are compact *i.e.*, can always be made on the basis of a finite set of formulas, whereas learning would be too restrictive if it always meant arriving at a definite conclusion on the basis of a finite set of data and background knowledge. So implementing a

* Eric Martin is supported by the Australian Research Council Grant A49803051. Frank Stephan is supported by the Deutsche Forschungsgemeinschaft (DFG) Heisenberg Grant Ste 967/1-1.

learning strategy in Prolog requires the use of heuristics that cannot be viewed as a description of the learning problem, since the latter is in essence nondeductive. Unless, if learning can legitimately be viewed as a particular kind of logical, nondeductive inference.

Indeed, a generalized notion of logical consequence can be defined that, given a theory T, yields a hierarchy of generalized logical consequences of T by alternating deductive (or compact) inferences and inductive (or 'weakly compact') inferences. This notion of generalized logical consequence is actually a function of a number of parameters, one of them being a class $\mathcal{W}$ of possible worlds, or intended interpretations. In most applications, a natural choice for $\mathcal{W}$ is the class of Henkin (if equality is allowed) or Herbrand (if equality is not allowed) structures.[3] In classical logic, where $\mathcal{W}$ is the class of all structures, each level of the hierarchy of (generalized) logical consequences collapses to the first level *i.e.*, the level of deductive inferences. But when $\mathcal{W}$ is equal to the class of Henkin or Herbrand structures, a true hierarchy of generalized logical consequences results. It turns out that under natural assumptions, identification in the limit can be characterized as the inference (in the limit) of a generalized logical consequence that belongs to the third level of the hierarchy, *i.e.*, the level of inductive inferences followed by deductive inferences. So a learning problem can be expressed in purely logical terms, and the ideal of programming in logic can be extended to applications such as learning that are beyond the scope of classical logic.

We proceed as follows. We briefly sketch some fundamental concepts of the generalized logic. We describe RichProlog, which is a natural enrichment of Prolog based on a very particular instance of the generalized logic that is capable of answering Σ_2 and Π_1 queries in addition to the usual Σ_1 queries that can answered by Prolog. Finally, we show how RichProlog can be used to learn nonerasing pattern languages from positive data.

The motivation for development of systems capable of answering Σ_2 queries is obvious from the perspective of discovering more sophisticated knowledge. Consider the hypothetical scenario of finding a vaccine for a virus. Let variable y range over different instantiations of a virus. Let $P(x, y)$ describe the property that x "disables" y. P may be a complex relation based on certain geometric and chemical properties between x and y. Then, the search for a vaccine is an answer to the Σ_2 query: $\exists x \forall y P(x, y)$. While this is a hypothetical scenario, a number of problems in drug design, e.g., *pharmacophore identification*, can be expressed as Σ_2 queries.

2 Logical foundation

2.1 Generalized logical consequence

Denote by S a vocabulary without equality and by $\mathcal{L}$ the set of first-order S-formulas, referred to more simply as *formulas*. Closed formulas will be called

[3] A Henkin structure consists of individuals each of which interprets a closed term. A Herbrand structure is a Henkin structure such that distinct closed terms are interpreted by distinct individuals.

sentences; atomic formulas or their negations will be called *basic* formulas. We refer to sets of formulas as *theories*. Denote by $\mathcal{E}$ a set of formulas, called set of *possible evidence*. Denote by $\mathcal{W}$ a class of S-structures, called class of *possible worlds*. For all $\mathfrak{M} \in \mathcal{W}$, the $\mathcal{E}$-*diagram of* $\mathfrak{M}$, denoted $D_{\mathcal{E}}(\mathfrak{M})$, is the set of all members of $\mathcal{E}$ true in $\mathfrak{M}$. Here is a typical scenario that could be described as a *paradigm* of Formal Learning Theory. (See [5, 9, 6] for descriptions and investigations of such paradigms.) A member of $\mathcal{W}$, say $\mathfrak{M}$, is chosen. A *learner* f is presented with every initial segment of an infinite enumeration of the diagram of $\mathfrak{M}$ and reacts by outputting members of $\mathcal{L}$.[4] More formally:

Definition 1. *A* learner *is a mapping*[5] *from* $\mathcal{E}^\star$ *into* $2^{\mathcal{L}}$.

Suppose for instance that S consists of a constant $\overline{0}$, a unary function symbol s, a unary predicate P, and a binary predicate R. Given $n \in \mathbb{N}$, denote by $\overline{n}$ the term obtained from $\overline{0}$ by n applications of s. Set $\mathcal{E} = \{P(\overline{n}) \mid n \in \mathbb{N}\}$. Assume that $\mathcal{W}$ is the class of Herbrand S-structures $\mathfrak{M}$ such that:

- for all $m, n \in \mathbb{N}$, $\mathfrak{M} \models R(\overline{m}, \overline{n})$ iff $m \leq n$;
- $\mathfrak{M} \models P(\overline{n})$ for finitely many $n \in \mathbb{N}$.

Choose for $\mathfrak{M}$ the (unique) member of $\mathcal{W}$ such that for all $n \in \mathbb{N}$, $\mathfrak{M} \models P(\overline{n})$ iff $n \leq 10$. The learner f could be presented with every initial segment of the initial sequence $e = (P(\overline{0}), P(\overline{1}), \ldots P(\overline{10}), P(\overline{0}), P(\overline{1}), \ldots P(\overline{10}), \ldots)$. We might expect f to be able to *discover in the limit* that the formula $\psi = P(\overline{10}) \wedge \forall y(R(s(\overline{10}), y) \rightarrow \neg P(y))$ is true in $\mathfrak{M}$. That is, faced with longer and longer initial segments of e, the learner f should be able to stabilize its outputs to ψ. More formally:

Definition 2. *A learner f* identifies a formula φ in the limit in $\mathcal{W}$ *just in case for all $\mathfrak{M} \in \mathcal{W}$ and infinite enumerations $(e_0, e_1, e_2 \ldots)$ of members of $D_{\mathcal{E}}(\mathfrak{M})$ where every member of $D_{\mathcal{E}}(\mathfrak{M})$ occurs at least once, $\mathfrak{M} \models \varphi$ iff $\varphi \in f((e_0, \ldots, e_k))$ for all but finitely many $k \in \mathbb{N}$.*

Let theory $X = \{R(\overline{m}, \overline{n}) \mid m \leq n\} \cup \{P(\overline{n}) \mid n \leq 10\}$. Now note that ψ is not a logical consequence of X. Indeed, the requirement—*every* possible evidence true in the underlying world will eventually appear in the enumeration—cannot be expressed in first-order logic. Moreover, ψ is not even a logical consequence of theory $Y = X \cup \{\neg P(\overline{n}) \mid n > 10\}$ because of the *nonstandard* models of Y, models of Y some of whose individuals are not interpreted by a closed term. Still, ψ can be viewed as a 'generalized logical consequence' of X in a 'generalized logic' that is not as rigid as classical first-order logic, because:

- it does not force us to accept structures we do not want to or have good reasons not to consider as possible interpretations;

[4] To be extremely precise, an extra symbol $\sharp$ can also appear in such enumerations, meaning 'no datum now.' This becomes necessary if no member of $\mathcal{E}$ is true in $\mathfrak{M}$.

[5] For the purpose of introducing the basic concepts of learning theory, whether this mapping should necessarily be computable is inessential. Given a set X, $X^\star$ denotes the set of finite sequences of members of X, and 2^X the set of all subsets of X.

– it has a minimality principle that captures the requirement that if some possible evidence χ does not belong to a theory T, then $\neg\chi$ should be considered to be true in every intended interpretation of T.

The second condition is obviously closely related to circumscription and the closed world assumption [11, 8]. The notion of generalized logical consequence that satisfies both conditions above is then formally defined via the following two definitions, where $\subset$ denotes strict inclusion.

Definition 3. *Let $T \subseteq \mathcal{L}$ and a structure $\mathfrak{M}$ be given. We say that $\mathfrak{M}$ is an* $\mathcal{E}$-minimal model of T in $\mathcal{W}$ *iff $\mathfrak{M}$ is a model of T in $\mathcal{W}$ and for all models $\mathfrak{N}$ of T in $\mathcal{W}$, $D_{\mathcal{E}}(\mathfrak{N}) \not\subset D_{\mathcal{E}}(\mathfrak{M})$.*

Definition 4. *Given $T \subseteq \mathcal{L}$ and $\varphi \in \mathcal{L}$, we say that φ is an* $\mathcal{E}$-minimal logical consequence of T in $\mathcal{W}$, *and we write $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$, iff every $\mathcal{E}$-minimal model of T in $\mathcal{W}$ is a model of φ.*

Definition 4 is a particular case of the notion of *preferential satisfaction* introduced in [13]. Getting back to the example above (where the values of $\mathcal{W}$, $\mathcal{E}$, X and ψ have been fixed), it is clear that ψ is an $\mathcal{E}$-minimal logical consequence of X in $\mathcal{W}$. More informally, we say that ψ is a *generalized logical consequence of X*. We refer the reader to [10] for a detailed development of the notion of generalized logical consequence.

2.2 From generalized logic to generalized logic programming

Definitions 3 and 4 generalize basic concepts of the theory of Logic Programming. Indeed, assume that $\mathcal{W}$ and $\mathcal{E}$ are defined as above. If T is a set of *rules* that contains $\{R(\overline{m}, \overline{n}) \mid m \leq n\}$ and if φ is an existential query, then 'φ is a generalized logical consequence of T' can be paraphrased as 'the (unique) minimal Herbrand model of T is a model of φ,' which we all know is equivalent to: φ is a logical consequence of T [4]. When the theory of Logic Programming is concerned with more general kinds of logic programs or more general kinds of queries, the equivalence fails. Since the focus is still on the classical notion of logical consequence, the intended interpretations can no longer be limited to the class of Herbrand structures. Our framework is the exact dual of this approach. We are concerned with more general kinds of logic programs and more general kinds of queries. Since Herbrand models are intended interpretations, the classical notion of logical consequence is inadequate. Consider the same example again. Representing the task of the learner f as a generalized logic program and a generalized query cannot be done directly in the realm of classical logic, because the task of f is not to discover that some formula is a logical consequence of some theory, but is to discover that some formula is a generalized logical consequence of some theory. Hence, it should be possible to represent the task of f directly, naturally and declaratively as a generalized logic program and a generalized query in the realm of the generalized logic we have introduced. More precisely, no generalized query would represent the task of f better than:

$$\varphi = \exists x \forall y (P(x) \wedge (R(s(\overline{x}), y) \rightarrow \neg P(y)).$$

Given the right generalized logic program T, we expect an interpreter to be able to prove that $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$, and as a side effect, that the formula obtained from φ by removing the existential quantifier and instantiating x with $\overline{10}$, is also a generalized logical consequence of T. In other words, we expect that it is possible to *compute* the least natural number m such that for all $n \in \mathbb{N}$, $\neg P(\overline{n})$ is true in the underlying world if and only if n is greater than m.

We will not formalize here how we define *hierarchies* of generalized logical consequences, but we explain their fundamental features. (See [10] for a formal treatment.) Denote by $\mathcal{A}$ a set of formulas that contains $\mathcal{E}$, called set of *possible axioms* (for instance, $\mathcal{A}$ can be defined as the set of rules). Call *possible theory* any set of the form $D_{\mathcal{E}}(\mathfrak{M}) \cup X$, where $\mathfrak{M}$ is a possible world and $X \subset \mathcal{A}$ is a set of possible axioms that are true in $\mathfrak{M}$. A hierarchy of generalized logical consequences of T can be defined for every possible theory T. Such a hierarchy reflects the complexity of generalized logical consequence. Basically, the higher a formula φ occurs in the hierarchy built over a possible theory T, the more difficult is the task of discovering that $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$. The first level of the hierarchy, called the Σ_1 level, corresponds to *deductive inference*: from a finite subset of T, it is possible to conclude with certainty that φ is a generalized logical consequence of T. The next level, called the Π_1 level, corresponds to *inductive inference*: from a finite subset of T, it is possible to believe that φ is a generalized logical consequence of T, since some finite subset of T can *refute* this belief with certainty in case $T \not\models^{\mathcal{E}}_{\mathcal{W}} \varphi$. The very principles that define the Σ_1 and Π_1 levels of the hierarchies can be iterated to define higher levels, starting with the Σ_2 level, followed by the Π_2 level, then the Σ_3 level, then the Π_3 level, etc. Under some assumptions, it is possible to relate the syntactic complexity of a formula with its location in the hierarchy of generalized logical consequences, as shown by the following proposition.

Proposition 1. *Suppose that $\mathcal{W}$ is a set of Henkin structures, $\mathcal{E}$ contains all basic sentences, and $\mathcal{A} = \mathcal{E}$. Let $n > 0$ be given. For every Σ_n (respect. Π_n) formula φ and possible theory T, if $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$ then φ belongs to the Σ_n (respect. Π_n) level of the hierarchy of generalized logical consequences of T.*

Proposition 1 is just one of many propositions that describe the structure of the hierarchies of generalized logical consequences built over possible theories, under various assumptions. It can be shown that there exists a learner that identifies in the limit a formula φ (in the sense of Definition 2) iff φ belongs to the Σ_2 level of the hierarchy of generalized logical consequences of T, for every possible theory T such that $T \models^{\mathcal{E}}_{\mathcal{W}} \varphi$—see [10]. So by Proposition 1, under some assumptions on $\mathcal{W}$, $\mathcal{E}$ and $\mathcal{A}$, the class of formulas that can be identified in the limit is precisely the class of Σ_2 formulas. But Σ_2 formulas can represent *learning problems.* For instance, the problem of learning a law from observed data can be expressed by the Σ_2 statement: 'there exists a law ℓ such that for all possible data d, ℓ predicts d iff d is among the data that are eventually observed.' Solving the learning problem is then reduced to computing a witness for ℓ that represents the law to be learned. When the set $\mathcal{E}$ of possible observations is not closed under negation, just a subclass of the class of all Σ_2 formulas can be identified in the limit. But rather than developing the logical framework and the connections with learning theory, we will devote the remaining part of this paper to a description of RichProlog. RichProlog is a natural extension of Prolog that can be used to infer that a sentence of the form $\exists\overline{x}\forall\overline{y}\varphi$, where φ is quantifier free, is a generalized logical consequence of a possible theory T encompassing background knowledge and observations. The result of the computation is a sequence of terms $\overline{t}$ such that $T \models^{\mathcal{E}}_{\mathcal{W}} \forall\overline{y}\varphi[\overline{t}/\overline{x}]$.[6] Solving a learning problem amounts to computing $\overline{t}$.

3 RichProlog

3.1 The basic strategy

From now on we assume that $\mathcal{W}$ is a set of Herbrand structures, while $\mathcal{E}$ is a set of atomic sentences. This is the natural choice when RichProlog is used to learn from positive data only. We denote by $\widehat{\mathcal{E}}$ the set of members of $\mathcal{E}$ and their negations. We assume that the set $\mathcal{A}$ of possible axioms is built from some set A of atomic formulas, none of which has a member of $\mathcal{E}$ as an instance, with the following property:

> $(*)$ every member of $\mathcal{A} \setminus \mathcal{E}$ is of the form $\alpha_1 \wedge \ldots \wedge \alpha_p \rightarrow \alpha_0$ (also represented as $\alpha_0 \leftarrow \alpha_1 \wedge \ldots \wedge \alpha_p$, or as α_0 :- $\alpha_1, \ldots, \alpha_p$) where $p \in \mathbb{N}$, α_0 is a member of A, and $\alpha_1, \ldots, \alpha_p$ are members of $A \cup \widehat{\mathcal{E}}$.

We call members of $\mathcal{A}$ *generalized clauses*. Remember that a possible theory is a set of the form $D_{\mathcal{E}}(\mathfrak{M}) \cup X$, where X is a subset of the set of possible axioms $\mathcal{A}$ all of whose members are true in $\mathfrak{M}$. We call *generalized logic program* a possible theory of the form $D_{\mathcal{E}}(\mathfrak{M}) \cup X$ where X is a finite set of possible axioms (*i.e.*, generalized clauses)—but $D_{\mathcal{E}}(\mathfrak{M})$ can obviously be infinite. The aim is to make RichProlog show that a Σ_2 formula φ is a generalized logical consequence of a generalized logic program $D_{\mathcal{E}}(\mathfrak{M}) \cup X$, on the basis of sets of the form $D \cup X$ where D is a finite subset of $D_{\mathcal{E}}(\mathfrak{M})$. Larger and larger subsets D of $D_{\mathcal{E}}(\mathfrak{M})$ will be provided to the system (which corresponds to making more and more observations). RichProlog has to infer correctly that φ is a generalized logical consequence of $X \cup D_{\mathcal{E}}(\mathfrak{M})$ when D is large enough and produce the right witnesses for the existentially quantified variables in φ. Note that for all possible theories T and for any possible evidence $\chi \in \mathcal{E}$, either $T \models^{\mathcal{E}}_{\mathcal{W}} \chi$ or $T \models^{\mathcal{E}}_{\mathcal{W}} \neg\chi$. In the first case, χ belongs to T and will eventually be provided to the system. In the second case, negation as failure correctly infers $\neg\chi$ from any finite subset of $X \cup D_{\mathcal{E}}(\mathfrak{M})$. Hence negation as failure represents 'true negation' w.r.t. the notion of generalized logical consequence. Weaker assumptions on $\mathcal{E}$ and $\mathcal{A}$ are possible and currently under investigation but in this paper, we limit the discussion to generalized clauses as defined above. Given a generalized logic program T and an atomic formula ψ all of whose free variables occur in the disjoint sequences of

[6] Given a formula ψ, $n \in \mathbb{N}$, a sequence $\overline{x} = (x_0, \ldots, x_n)$ of distinct variables, and a sequence $\overline{t} = (t_0, \ldots, t_n)$ of terms, $\psi[\overline{t}/\overline{x}]$ denotes the result of simultaneously substituting in ψ every occurrence of x_i by t_i, for all $i \leq n$.

variables $\overline{x}$ and $\overline{y}$, RichProlog will determine that $\exists\overline{x}\forall\overline{y}\psi$ is a generalized logical consequence of T whenever this is indeed the case. Moreover, when $\exists\overline{x}\forall\overline{y}\psi$ is a generalized logical consequence of T, RichProlog will output a sequence of terms $\overline{t}$ of the same length as $\overline{x}$, a witness for $\exists\overline{x}\forall\overline{y}\psi$, such that $T \models^{\mathcal{E}}_{\mathcal{W}} \forall\overline{y}\psi[\overline{t}/\overline{x}]$. More complex Σ_2 queries can also be tackled, as will be seen in the Section 3.3. Usually T is infinite (because the $\mathcal{E}$-diagram of a possible world is infinite), and RichProlog's outputs are correct *in the limit* that is, from the time when a large enough finite subset of T is available. For the kind of application that will be discussed in this paper, RichProlog's search strategy proceeds in the following two stages.

Stage 1: Choose $n \in \mathbb{N}$ and sequences of terms $\overline{t_1}, \ldots, \overline{t_n}$ of the same length as $\overline{y}$, and find a witness $\overline{t}$ for the query $\exists\overline{x}(\psi(\overline{x}, \overline{t_1}) \wedge \ldots \wedge \psi(\overline{x}, \overline{t_n}))$?

Stage 2: Try to *refute* $\forall\overline{y}\psi(\overline{t}, \overline{y})$ *i.e.*, try to find a witness for $\exists\overline{y}\neg\psi(\overline{t}, \overline{y})$. If no witness is found, *i.e.*, if $\forall\overline{y}\psi(\overline{t}, \overline{y})$ is *validated* then output $\overline{t}$; otherwise backtrack to Stage 1 and find another witness for $\exists\overline{x}(\psi(\overline{x}, \overline{t_1}) \wedge \ldots \wedge \psi(\overline{x}, \overline{t_n}))$?

A completeness result would show that RichProlog's outputs stabilize to a correct witness $\overline{t}$ as soon as the finite fragment of T being dealt with is large enough, and as soon as enough sequences of terms $\overline{t_1}, \ldots, \overline{t_n}$ have been chosen during Stage 1. We think that the completeness results we have obtained so far can be extended to larger classes of generalized logic programs and Σ_2 queries, so we do not address the issue in this paper. Note that backtracking takes place withing Stage 1, within Stage 2, and from Stage 2 to Stage 1. RichProlog needs to validate the witness output every time Stage 1 is exited. It involves trying to refute a Π_1 sentence, hence trying to prove a Σ_1 sentence ξ. Note that negation has to occur either in the matrix of ξ or in the matrix of the initial Σ_2 query. A solution that works in many cases is to consider only Σ_2 queries in which the negated atoms are members of $\mathcal{E}$ only, and to use in Stage 2 a Σ_1 query ξ' that is logically equivalent to ξ modulo the background knowledge, but such that the negated atoms in ξ' are also members of $\mathcal{E}$ only. (As noticed above, the definition of a generalized clause and the notion of generalized logical consequence are such that negation applied to members of $\mathcal{E}$ can be trivially handled by negation as failure.) We will briefly return to the issue of negation in Section 3.4.

The search performed during Stage 1 is not very efficient. Indeed, instead of instantiating the universally quantified variables of the initial query by $\overline{t_i}$, $1 \leq i \leq n$, we could *target* $\overline{t_i}$, solving the query $\exists\overline{x}\exists\overline{y}\psi(\overline{x}, \overline{y})$? and making sure that the witness $(\overline{t}, \overline{t'})$ for this query is such that $\psi(\overline{t}, \overline{t'})$ is an instance of $\psi(\overline{t}, \overline{t_i})$ (equivalently, such that $\overline{t'}$ is an instance of $\overline{t_i}$). If $\overline{t'}$ contains variables, then some of the sequences of terms $\overline{t_{i+1}}, \ldots, \overline{t_n}$ could be instances of $\overline{t'}$ and 'skipped.' On the other hand, when it is known that Stage 1 is always exited with a witness $(\overline{t}, \overline{t'})$ for the query $\exists\overline{x}\exists\overline{y}\psi(\overline{x}, \overline{y})$? such that $\overline{t'}$ does not contain variables, the previous search strategy amounts to the one described above. The application to be presented in Section 4 has this property, hence we have sacrificed generality for simplicity and coherence. The reader interested in the more complex search strategy implemented by RichProlog during Stage 1 will find it described by the algorithm given in Section 3.4.

3.2 An illustration

As described in the previous section, RichProlog proposes witnesses using a strategy that consists of two stages, where each stage is a search for solution to some Σ_1 query. The following illustration is presented with the purpose of exemplifying how a search tree for RichProlog is structured. It differs from the structure of a search tree for Prolog due to the confirmation process of Stage 2. The reader should note that the following illustration is not a search tree associated with an actual program execution; hence the assignments to variables are coherent and plausible, but arbitrary. Space constraints prevent us from presenting an actual example. We consider the example where the

```
              X=X1, Y=Y1
1.        [p(X1,Y1,a,a)?]

           X1=f(X2), Y1=g(Y2)
2.      [p(f(X2),g(Y2),a,b)?]

            X2=g(X3), Y2=b                        X2=b, Y2=a
3a.    [p(f(g(X3)),g(b),b,a)?]      3b.     [p(f(b),g(a),b,a)?]

                X3=a
4a.   [p(f(g(a)),g(b),a,f(b))?]     4b.   [p(f(b),g(a),a,f(b))?]
```

Hypothesis: `X=f(g(a)), Y=g(b)`
VALIDATION
$\boxed{\neg\ \mathtt{p(f(g(a)),g(b),Z,W)?}}$

Validation fails

Hypothesis: `X=f(b), Y=g(a)`
VALIDATION
$\boxed{\neg\ \mathtt{p(f(b),g(a),Z,W)?}}$

Fig. 1. An example of search tree for the query $\exists \mathtt{X}\ \mathtt{Y} \forall \mathtt{Z}\ \mathtt{W}\ \mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})$?

query is $\exists\mathtt{X}\exists\mathtt{Y}\forall\mathtt{Z}\forall\mathtt{W}\,\mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{Z},\mathtt{W})$?. Assume that we decide exiting Stage 1 when a common witness for $\exists\mathtt{X}\exists\mathtt{Y}\,\mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{a},\mathtt{a})$?, $\exists\mathtt{X}\exists\mathtt{Y}\,\mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{a},\mathtt{b})$?, $\exists\mathtt{X}\exists\mathtt{Y}\,\mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{b},\mathtt{a})$?, $\exists\mathtt{X}\exists\mathtt{Y}\,\mathtt{p}(\mathtt{X},\mathtt{Y},\mathtt{a},\mathtt{f}(\mathtt{b}))$? is found. A possible search tree for this query is depicted in Figure 1. At the beginning of the search, the pair of variables `(Z,W)` is assigned the first pair of closed terms `(a,a)` (step 1.). The variables `Z` and `W` are subsequently assigned the values of the other pairs of closed terms in the following iterations: `(a,b)` at step 2., `(b,a)` at step 3., `(a,f(b))` at step 4. The instances of the existentially quantified variables become more and more specific since they have to cover more and more instances of the universally quantified variables: they become `(X1,Y1)` at step 1., `(f(X2),g(Y2))` at step 2., `(f(g(X3)),g(b))` at step 3a., and `(f(g(a)),g(b))` at step 4a. Following step 4a., enough instances of the universally quantified variables have been covered, and Stage 1 of the search strategy is exited with a witness equal to `(f(g(a)),g(b))`. The execution now begins to validate $\forall \mathtt{Z}, \mathtt{W}\ \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}), \mathtt{Z}, \mathtt{W})$. This is done by trying to find a counterexample, *i.e.*, carrying out the search for a possible solution to the query $\exists \mathtt{Z}, \mathtt{W} \neg \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}), \mathtt{Z}, \mathtt{W})$?. Suppose that this search actually succeeds, *i.e.*: some witness for $\exists \mathtt{Z}, \mathtt{W} \neg \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}), \mathtt{Z}, \mathtt{W})$? is found. Then $(\mathtt{f}(\mathtt{g}(\mathtt{a})), \mathtt{g}(\mathtt{b}))$ is discarded and the search backtracks to step 4. If the search for a new witness for the query $\exists \mathtt{X3}\, \mathtt{p}(\mathtt{f}(\mathtt{g}(\mathtt{X3})), \mathtt{g}(\mathtt{b}), \mathtt{a}, \mathtt{b})$? fails, then the whole search backtracks to step 3. In Figure 2, it is assumed that a witness for $\exists \mathtt{X2}, \mathtt{Y2}\, \mathtt{p}(\mathtt{f}(\mathtt{X2}), \mathtt{g}(\mathtt{Y2}), \mathtt{b}, \mathtt{a})$? is found (step 3b.) that is also a witness for the next query (step 4b.). Enough instances of the universally quantified variables have been covered, and Stage 1 of the search strategy is exited with a witness equal to `(f(b),g(a))`. The execution now begins to validate $\forall \mathtt{Z}, \mathtt{W}\ \mathtt{p}(\mathtt{f}(\mathtt{b}), \mathtt{g}(\mathtt{a}), \mathtt{Z}, \mathtt{W})$. No counterexample is found and $(f(b), g(a))$ is output (possibly by stopping the unsuccessful search for a counterexample).

3.3 Complex queries

In Section 3.1, we have considered Σ_2 queries whose matrix is an atomic formula. We now examine how to deal with more complex queries, transforming them into queries whose matrix is an atomic formula modulo an extension of T with a set of generalized clauses. Remember from the previous section how generalized clauses have been defined from the set denoted $\widehat{\mathcal{E}}$ and a set A of atomic formulas. Here we consider *generalized Σ_2 queries* defined as Σ_2 sentences whose matrix is built from $A \cup \widehat{\mathcal{E}}$ using disjunction and conjunction only. Basically, a new predicate symbol is introduced for each inner node in the parse tree of the matrix of the query. These symbols are not part of the vocabulary S; they are new symbols that only appear in a program derived from T—the initial generalized program—together with a particular generalized Σ_2 query. Each of these new predicate symbols must be of arity equal to the total number of (both the existentially quantified and the universally quantified) variables in the query. The query's matrix itself is replaced by the new predicate assigned to the root of the parse tree. Figure 2 depicts the parse tree for the matrix of the query $\varphi = \exists \mathtt{X} \exists \mathtt{Y} \forall \mathtt{Z} \psi$ where $\psi = \mathtt{p}_1(\mathtt{X}, \mathtt{Y}) \wedge [\neg \mathtt{p}_2(\mathtt{Z}) \vee (\mathtt{p}_3(\mathtt{X}, \mathtt{Y}) \wedge \mathtt{p}_4(\mathtt{Y}) \wedge (\neg \mathtt{p}_2(\mathtt{Y}) \vee \neg \mathtt{p}_3(\mathtt{X}, \mathtt{Z})))]$. Note that it is not necessary to introduce new predicate symbols for the leaves of the parse tree, and that the leaves can be labelled with atomic as well as with negations of atomic formulas which in the latter case, are necessarily negations of members of $\mathcal{E}$. In this example, four new predicates have been created. Correspondingly, some generalized clauses will be created in the transformation of a particular generalized Σ_2 query, and will be used in the proof for that query only. In the following, 'the predicate symbol at node N' will denote the new predicate symbol introduced at N if N is an inner node, and the predicate symbol (from S) when N is a leaf. Basically, if an inner node is an *and-node*, then one generalized clause is created whose head is the predicate symbol at that node, and whose body is the conjunction of the predicate symbols at the children of the node. On the other hand, if the inner node is an *or-node*, then for each child of the node, one generalized clause is added to the program. The head of these generalized clauses is the predicate symbol at the parent node, while their body is the pred-

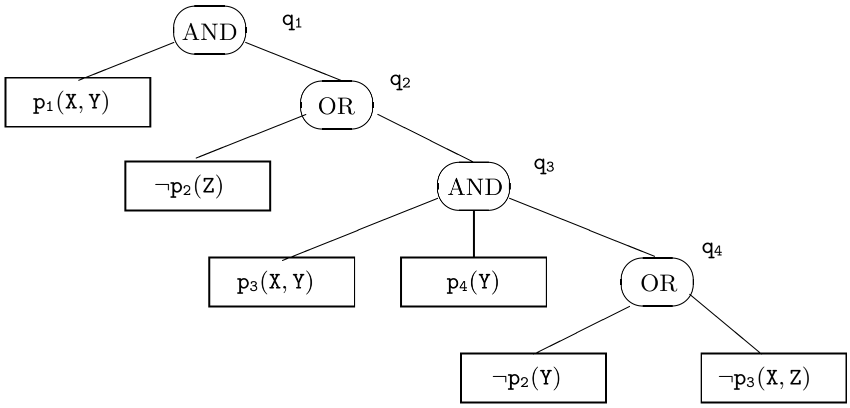

Fig. 2. Parse tree for $\mathtt{p}_1(\mathtt{X},\mathtt{Y}) \wedge [\neg\mathtt{p}_2(\mathtt{Z}) \vee (\mathtt{p}_3(\mathtt{X},\mathtt{Y}) \wedge \mathtt{p}_4(\mathtt{Y}) \wedge (\neg\mathtt{p}_2(\mathtt{Y}) \vee \neg\mathtt{p}_3(\mathtt{X},\mathtt{Z})))]$

icate symbol at the child node. In case of the previous example, the generalized clauses are given below and the query will be transformed into $\exists\mathtt{X}\exists\mathtt{Y}\forall\mathtt{Z}\ \mathtt{q}_1(\mathtt{X},\mathtt{Y},\mathtt{Z})$.

$$\begin{aligned}
&\mathtt{q}_1(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}_1(\mathtt{X},\mathtt{Y}), \mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z})\\
&\mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \neg\mathtt{p}_2(\mathtt{Z})\\
&\mathtt{q}_2(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{q}_3(\mathtt{X},\mathtt{Y},\mathtt{Z})\\
&\mathtt{q}_3(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \mathtt{p}_3(\mathtt{X},\mathtt{Y}), \mathtt{p}_4(\mathtt{Y}), \mathtt{q}_4(\mathtt{X},\mathtt{Y},\mathtt{Z})\\
&\mathtt{q}_4(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \neg\mathtt{p}_2(\mathtt{Y})\\
&\mathtt{q}_4(\mathtt{X},\mathtt{Y},\mathtt{Z}) \leftarrow \neg\mathtt{p}_3(\mathtt{X},\mathtt{Z})
\end{aligned}$$

3.4 Further remarks

As has been explained at the end of Section 3.1, RichProlog's search strategy for Stage 1 is more complex than the simple approach that has been described above. We now give, without comment for lack of space, the nondeterministic version of the algorithm used by RichProlog for Stage 1. Let μ denote a measure over the set of all closed terms, extended to a measure over the set of all terms by $\mu(t) = \mu(\{t' \mid t' \text{ is a closed instance of } t\})$. The m-product of μ is also denoted μ. We assume that some enumeration of all m-tuples of closed terms is given. Figure 3 describes the algorithm where δ represents a threshold that used together with μ, plays the role of the number n of atoms in the query of Stage 1. Remember that starting from a generalized Σ_2 query, the validation process requires trying to prove a Σ_1 query ξ, where, apart from trivial cases, negations of atomic sentences not in $\mathcal{E}$ occur. We have defined generalized logic programs and generalized Σ_2 queries so as to bypass the issue of true negation, but this issue seems to pop up inevitably at Stage 2 of the basic strategy. Still, in this paper and in most applications, we can avoid having to deal with true negation even for the validation process: it suffices that the formula $\exists\overline{y}\neg\psi(\overline{t},\overline{y})$ be logically equivalent to a Σ_1 query whose matrix is like the matrix of a generalized Σ_2 query: obtained

Input: A finite logic program T, a sentence of the form $\exists x_1 \dots \exists x_n \forall y_1 \dots \forall y_m \psi$ where ψ is atomic, and a rational number δ in $(0, 1)$.
Output: A witness for $\exists x_1 \dots \exists x_n \forall y_1 \dots \forall y_m \psi$.

Initialize Y to $\emptyset$.
Initialize $(t_1, \dots, t_n)$ to $(x_1, \dots, x_n)$.
While $\mu(Y) < \delta$ *do*

1. $Res = \{\psi\}$, $k = 0$.
2. $(t'_1, \dots, t'_m)$ = the first member of the enumeration of m-tuples of closed terms that is not an instance of a member of Y.
3. *While* $Res \neq \emptyset$ *do*
   - 3.1. Choose ρ in Res and (renamed) clause $\alpha_0 \leftarrow \alpha_1 \dots \alpha_p$ in T such that
     - ρ and α_0 unify with mgu θ_k;
     - $(t'_1, \dots, t'_m)$ is an instance of $(y_1, \dots, y_m)\theta_0 \dots \theta_k$.

     (If no such ρ and $\alpha_0, \dots, \alpha_p$ exist, exit 3.)
   - 3.2 Replace ρ by $\alpha_1, \dots, \alpha_p$ in Res.
   - 3.3 Apply θ_k to Res.
   - 3.4 $k = k + 1$.
4. *If* $Res \neq \emptyset$ *then* output *no*.
5. *Else* let θ be a most general such that:
   - no variable occurs both in $(t_1, \dots, t_n)\theta_0 \dots \theta_k\theta$ and $(y_1, \dots, y_m)\theta_0 \dots \theta_k\theta$;
   - $(t'_1, \dots, t'_m)$ is an instance of $(y_1, \dots, y_m)\theta_0 \dots \theta_k\theta$.
6. $(t_1, \dots, t_n) = (t_1, \dots, t_n)\theta_0 \dots \theta_k\theta$.
7. $Y = Y \cup \{(y_1, \dots, y_m)\theta_0 \dots \theta_k\theta\}$.

Output $(t_1, \dots, t_n)$.

Fig. 3. A nondeterministic algorithm for a more efficient search during Stage 1

from $A \cup \widehat{\mathcal{E}}$ using disjunction and conjunction only. This will be illustrated in the application to be examined now.

4 An application

4.1 Description of the problem

In this section, we show how RichProlog can be used to solve the problem of learning nonerasing pattern languages from positive data (see [1, 2]). Consider the alphabet $\{\mathtt{a}, \mathtt{b}\}$ and an infinite sequence of variables $\mathtt{V1}, \mathtt{V2}, \dots$. Define *words* as nonempty finite sequences over $\{\mathtt{a}, \mathtt{b}\}$, and *patterns* as nonempty finite sequences over $\{\mathtt{a}, \mathtt{b}, \mathtt{V1}, \mathtt{V2}, \dots\}$. Hence words are particular kinds of patterns. Given a pattern π, an *instance* of π is a word obtained from π by replacing all variables in π by words, with the same replacement for occurrences of the same variables. A word w and a pattern π are said to *match* iff w is an instance of π. Consider a learner f which is presented with every finite initial segment of a (clearly infinite) enumeration of all instances of π, for some arbitrary pattern π. The task of f is to discover π in the limit. The problem can be naturally cast in the logical framework as follows. Choose S such that constants and function symbols in S enable to represent a pattern by a closed term. Also put a predicate symbol P in S. A pattern π can then be represented by a Herbrand structure $\mathfrak{M}$ such that for all closed terms t, $\mathfrak{M} \models P(t)$ iff t represents a word that is an instance of π. Define the set $\mathcal{E}$ of possible evidence as $\{P(t) \mid \text{ closed term } t\}$. Remember that a possible theory T contains at least the $\mathcal{E}$-diagram of a possible world $\mathfrak{M}$. In this example, the $\mathcal{E}$-diagram of $\mathfrak{M}$ corresponds precisely to the set of words that are instances of the pattern represented by $\mathfrak{M}$. But T also contains a set X of possible axioms (which for RichProlog, must be a set of possible clauses) that are true in $\mathfrak{M}$. For this application, X can be defined independently of $\mathfrak{M}$, and will basically define the relationship between pattern and instances. Then we just have to ask a Σ_2 query whose intended meaning is: *does there exist a pattern π such that for all closed terms t that represent a word w, w is an instance of π iff $P(t)$ is observed*? This sentence will be a generalized logical consequence of $X \cup D_{\mathcal{E}}(\mathfrak{M})$ where $\mathfrak{M}$ is the structure that represents π. So in the limit, that is provided with enough possible evidence true in $\mathfrak{M}$, RichProlog outputs a witness t for the query such that t represents π.

4.2 Specifying the problem in RichProlog

A *datum* will refer to an arbitrary word, a *positive datum* to some instance of the pattern to be learned, and a *negative datum* to other words.

The hypothesis space The learner will have access to the hypothesis space (the set of patterns) by the ability to generate every pattern. Patterns of length N will be generated before patterns of lenght $N+1$. The rules for this generation are 1.,2.,6.–13. in Figure 4. Basically, the predicate `pattern(Pi,N)` is true iff `Pi` is a pattern of length `N`. Starting from a general pattern (constant `gen_pattern`) of lenght `N`, equal to `V1V2...VN` up to a renaming of variables, a variable `Vi`, $\texttt{1} \leq \texttt{i} \leq \texttt{N}$, can be replaced by `a` (thanks to the binary function symbol `subst_a`) or by `b` (thanks to the binary function symbol `subst_b`). Also given a variable `Vi`, $\texttt{1} \leq \texttt{i} \leq \texttt{N}$, a variable `Vj` with $\texttt{1} \leq \texttt{j} < \texttt{i}$ can be picked up (using the binary predicate symbol `select`), and `Vi` and `Vj` can be identified (thanks to the ternary function symbol `eq_var`).

The data The unary predicate symbol `p` is used to present positive data to the learner. Before asking a query, users should insert a number of statements corresponding to the data given to the learner. At any point, using negation as failure, the learner will consider any word which has not yet been presented (and maybe will never be) as not being an instance of the pattern to be learned. Obviously, some of these words will be presented later in time. For this application, this approach is not misleading, as will be explained below. For other applications involving learning from positive data only, negation as failure is not the right approach and true negation has to be dealt with in one way or another.

The learner The learner must be able to determine whether a pattern and a word match or not. The binary predicate symbol `match` is to be interpreted as a relation between a pattern and a word, true when both match. The learner's strategy is to examine the shortest pattern which matches every positive datum, and does not match any other. It can be easily verified that a pattern of length `N` will not match at least one of the negative data of length `N`, if there is any such word. Due to the limiting nature of the problem, if a pattern `Pi` of length `N` is to be learned, then eventually all positive data of length `N` will be given to the learner. At that point, its assertion about the negative data of length `N` will be correct. Considering some words greater than `N` as negative data might be wrong, but it does not affect the learning strategy: the learner needs to validate hypotheses using *only* words of length `N`. For the implementation of `match` we refer to Figure 4. Essentially, `match` only succeeds when the pattern and the words are of the same length and match.

The query We now consider the query that can be handled by RichProlog. The learner needs to hypothesize a pattern which matches all positive data, and does not match any word considered to be a negative datum. It does so by first generating possible hypotheses, and then trying to validate them. Generating a hypothesis can be done using the query:

$$\begin{aligned}\exists \mathtt{Pi},\mathtt{N}[\mathtt{pattern}(\mathtt{Pi},\mathtt{N}) \wedge \forall \mathtt{W}((&\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \mathtt{match}(\mathtt{Pi},\mathtt{W}) \rightarrow \mathtt{p}(\mathtt{W}))\wedge \\ &(\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \mathtt{mismatch}(\mathtt{Pi},\mathtt{W}) \rightarrow \neg\mathtt{p}(\mathtt{W})))]\end{aligned}$$

It is easy to express this formula as a Σ_2 sentence whose matrix is built from $\widehat{\mathcal{E}}$ and the predicates `pattern`, `length`, `distinct`, `match` and `mismatch` using disjunction and conjunction only, hence to express this formula as a generalized Σ_2 query. Then we transform this generalized Σ_2 query into a Σ_2 sentence whose matrix is an atomic formula, and we add to the program the following set of generalized clauses, as explained in Section 3.3.

```
 q(Pi,N,W) :- pattern(Pi,N), q1(Pi,N,W).
q1(Pi,N,W) :- p(W), q2(Pi,N,W).
q1(Pi,N,W) :- not p(W), q3(Pi,N,W).
q2(Pi,N,W) :- length(W,N1), distinct(N1,N).
q2(Pi,N,W) :- match(Pi,W).
q3(Pi,N,W) :- length(W,N1), distinct(N1,N).
q3(Pi,N,W) :- mismatch(Pi,W).
```

Validating the hypothesis involves the following refuting sentence, which aims at finding counterexamples for the proposed witness:

$$\begin{aligned}\exists \mathtt{W}[\neg\mathtt{pattern}(\mathtt{Pi},\mathtt{N}) \vee (&\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \mathtt{match}(\mathtt{Pi},\mathtt{W}) \wedge \neg\mathtt{p}(\mathtt{W}))\vee \\ &(\mathtt{length}(\mathtt{W},\mathtt{N}) \wedge \neg\mathtt{match}(\mathtt{Pi},\mathtt{W}) \wedge \mathtt{p}(\mathtt{W}))]\end{aligned}$$

Note that in the validation process, the pair `(Pi,N)` is assigned a pair of closed terms, thus $\neg$`pattern(Pi,N)` will always be false. Hence it can be removed from the validation query. It is then easy to express the refuting sentence as a Σ_1 sentence whose matrix is built from $\widehat{\mathcal{E}}$ and the predicates `length`, `match` and mismatch using disjunction and conjunction only. We transform again this Σ_1 sentence into a Σ_1 sentence whose matrix is an atomic formula, and we add to the program the following set of generalized clauses, as explained in Section 3.3.

```
r(Pi, N, W)  :-  p(W), length(W, N), mismatch(Pi, W).
r(Pi, N, W)  :-  not p(W), length(W, N), match(Pi, W).
```

```
1. numb(z).
2. numb(s(N)) :- numb(N).
3. distinct(s(N),z).
4. distinct(z,s(N)).
5. distinct(s(N),s(M)) :- distinct(N,M).
6. select(N,s(N)).
7. select(N,s(N)) :- select(N,N).
8. pattern(Pi,s(N)) :-
       numb(N), pattern(Pi,s(N)).
9. pattern(gen_pattern,z).
10. pattern(Pi,s(N)) :- pattern(Pi,N).
11. pattern(subst_a(Pi,s(N)),s(N)) :-
       pattern(Pi,N).
12. pattern(subst_b(Pi,s(N)),s(N)) :-
       pattern(Pi,N).
13. pattern(eq_var(Pi,s(M),s(N)),s(N)) :-
       select(M,N), pattern(Pi,N).
14. length(e,z).
15. length(a(W),s(N)) :- length(W,N).
16. length(b(W),s(N)) :- length(W,N).
17. symb_a(a(W),s(N)) :- length(W,N).
18. symb_a(a(W),N) :- symb_a(W,N).
19. symb_a(b(W),N) :- symb_a(W,N).
20. symb_b(b(W),s(N)) :- length(W,N).
21. symb_b(a(W),N) :- symb_b(W,N).
22. symb_b(b(W),N) :- symb_b(W,N).
23. match(gen_pattern,W).
24. match(subst_a(Pi,N),W) :-
       symb_a(W,N), match(Pi,W).
25. match(subst_b(Pi,N),W) :-
       symb_b(W,N), match(Pi,W).
26. match(eq_var(Pi,M,N),W) :-
       symb_a(W,N), symb_a(W,M),
         match(Pi,W).
27. match(eq_var(Pi,M,N),W) :-
       symb_b(W,N), symb_b(W,M),
         match(Pi,W).
28. mismatch(subst_a(Pi,N),W) :-
       symb_b(W,N).
29. mismatch(subst_b(Pi,N),W) :-
       symb_a(W,N).
30. mismatch(eq_var(Pi,M,N),W) :-
       symb_a(W,N), symb_b(W,M).
31. mismatch(eq_var(Pi,M,N),W) :-
       symb_b(W,N), symb_a(W,M).
32. mismatch(subst_a(Pi,N),W) :-
       mismatch(Pi,W).
33. mismatch(subst_b(Pi,N),W) :-
       mismatch(Pi,W).
34. mismatch(eq_var(Pi,N,M),W) :-
       mismatch(Pi,W).
```

Fig. 4. Logic program for learning pattern languages in the limit

Remarks We illustrate the behavior in the limit of the learner who has to learn the pattern V0V0V0. First aaa is presented. So a, b, aa, ab, ba, bb, bab, bbb, aba are treated (by negation as failure) as negative data. On the basis of aaa alone, the learner outputs the hypothesis V2aa. This hypothesis will be refuted by the (supposedly) negative datum baa. So the search will backtrack to Stage 1, and aaV0 will be the new witness. At some point, the learner is given the datum bbb. It will then output the hypothesis V1V1V0. Since bba does not belong to the list of positive data, it will be treated as a negative datum. The learner will then output the correct hypothesis V0V0V0. As can be verified, as soon as the correct hypothesis has been output, the learner will keep it in the face of any new data. RichProlog has been run on this program, which did not raise any efficiency issues. The purpose was just to demonstrate the feasibility of the approach. We are working on larger applications in order to investigate in depth how well RichPolog can compete against alternative approaches.

5 Conclusion

Generalized logic, generalized logic programming, and RichProlog can be seen as instances of a natural generalization, or 'lifting,' of classical logic, logic programming, and Prolog. This enrichment is based on a notion of generalized logical consequences that are not necessarily compact, but account for deductive inferences, inductive inferences, and inferences of higher complexity. In this paper, we gave an overview of the main aspects of the three components of this work, instead of a full account of some part of it. In particular, we have not touched upon the issue of the class of generalized logic programs and queries for which RichProlog provides a complete proof procedure. It should be noted that the notion of completeness of classical logic is inappropriate here, since RichProlog is designed to perform noncompact inferences: completeness has to be based on the concept of convergence in the limit. Our main aim was to show that RichProlog extends the ideal of declarative programming to applications such as learning. We argued that learning problems can also share the benefit of declarative programming, despite the fact that learning is by nature nondeductive.

References

1. Angluin, D.: *Finding patterns common to a set of strings.* Journal of Computer and System Sciences. **21** (1980) 46–62
2. Angluin, D.: *Inductive Inference of Formal Languages from Positive Data.* Information and Control. **45** (1980) 117–135
3. Apt, K., Bol, R.: *Logic Programming and Negation: A Survey.* Journal of Logic Programming. **19/20** (1994) 177–190
4. Doets, K.: *From Logic to Logic Programming.* The MIT Press. (1994)
5. Jain, S., Osherson, D., Royer, J., Sharma, A. *Systems that learn: An Introduction to Learning Theory, Second Edition.* The MIT Press. (1999)
6. Kelly, K.: *The Logic of Reliable Inquiry.* Oxford University Press. (1996).
7. Le, T.: *A general scheme for representing negative and quantified queries for deductive databases.* Proceedings of the First Internatinal Conference on Information and Knowledge Management. Baltimore, Maryland. (1992)
8. Lifschitz, V.: *Closed-World Databases and Circumscription.* Artificial Intelligence. **27** (1985) 229–235
9. Martin, E., Osherson, D.: *Elements of Scientific Inquiry.* The MIT Press. (1998)
10. Martin, E., Sharma, A., Stephan, F.: *A General Theory of Deduction, Induction, and Learning.* In Jantke, K., Shinohara, A.: Proceedings of the Fourth International Conference on Discovery Science. Springer-Verlag. (2001) 228–242.
11. Reiter, R.: *On Closed-World Data Bases.* In Gallaire, J., Minker, J., ed., Logic and Data Bases. Plenum Press. (1978) 55–76
12. Shepherdson, J.: *Negation in Logic Programming.* In Minker, J., ed., Foundations of Deductive databases and Logic Programming. Morgan Kaufmann. (1988) 19–88
13. Shoham, Y.: *Reasoning about change.* The MIT Press. (1988)