Incorporating a folding rule into inductive logic programming

David A. Rosenblueth
Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas
Universidad Nacional Autónoma de México
Apdo. 20-726, 01000 México D. F.
Mexico
drosenbl@servidor.unam.mx

Abstract

Many inductive logic programming systems have operators reorganizing the program so far inferred, such as the intra-construction operator of CIGOL. At the same time, there is a similar reorganizing operator, called the "folding rule," developed in program transformation. We argue that there are advantages in using an extended folding rule as a reorganizing operator for inductive-inference systems. Such an extended folding rule allows an inductive-inference system not only to recognize already-learned concepts, but also to increase the efficiently of execution of inferred programs.

1 Introduction

At first glance, it might seem that inductive logic programming (ILP) and logic program transformation (LPT) have opposite objectives. While ILP is interested in generalizing from examples, LPT is careful to preserve correctness. Nevertheless, there are ILP operators, like the intra-construction operator of CIGOL, that do not generalize, and instead reorganize the program so far inferred. This suggests looking at logic program transformation rules to obtain better nongeneralizing operators for ILP.

One LPT rule, called "folding," [Pettorossi and Proietti, 1998] is reminiscent of intra-construction, with some important differences. For example whereas intra-construction produces the definition of a new predicate (i.e. "concept"), folding receives as input such a definition, so that there is a separate, dedicated rule for creating a definition. The reason there is such a separate rule in LPT is that the derivation of efficiently executable programs is closely related to being able to fold using an already-existing definition.

Researchers have developed several reorganizing operators for ILP. Such operators can be found, for instance in the systems devised by Muggleton, Banerji, Ling, and Rouveirol and Puget. We have selected Muggleton's intra-construction as a representative of these operators for performing a comparison with folding. The main ideas of our comparison, however, should carry over to other similar operators.

Throughout, A and B denote atoms, and $\tilde{Q}$ and $\tilde{R}$ denote tuples of atoms (we will assume definite logic programs). Also, $\text{msg}(e_1, e_2)$ denotes the most specific generalization of expressions e_1 and e_2, and "$\simeq$" denotes equality up to variable renaming.

When used to fold two clauses, the *folding rule* infers (F) from (D_1), (E_1), (D_2), and (E_2):

$$\frac{\begin{array}{ll} B_1 \leftarrow \tilde{Q}_1 \quad (\text{D}_1) & B_2 \leftarrow \tilde{Q}_2 \quad (\text{D}_2) \\ A \leftarrow \tilde{Q}_1\theta_1, \tilde{R} \ (\text{E}_1) & A \leftarrow \tilde{Q}_2\theta_2, \tilde{R} \ (\text{E}_2) \end{array}}{A \leftarrow B, \tilde{R} \ \ (\text{F})}$$

where $B = B_1\theta_1 = B_2\theta_2$. In addition, there are some syntactic restrictions for soundness, for which we refer the reader to [Pettorossi and Proietti, 1998]. An application of folding replaces (E_1) and (E_2) by (F).

Observe that this rule has the limitation of requiring both clauses to be folded (E_1) and (E_2), to have the same head A and the same part of the body $\tilde{R}$ which is not an instance of the bodies of the definition (D_1) and (D_2).

On the other hand, folding has the advantage of allowing the clauses (D_1) and (D_2) to be drawn from a previous program [Pettorossi *et al.*, 1996].

To be able to perform a comparison with *intra-construction*, we will assume that intra-construction consists of two parts: first the "invention" of a predicate, and then the clause-reorganization proper. The second part would then be:

$$\frac{\begin{array}{ll} B_1 \leftarrow \quad (\text{D}_1) & B_2 \leftarrow \quad (\text{D}_2) \\ A_1 \leftarrow \tilde{R}_1 \ (\text{E}_1) & A_2 \leftarrow \tilde{R}_2 \ (\text{E}_2) \end{array}}{A \leftarrow B, \tilde{R} \ \ (\text{F})}$$

where

$$\begin{aligned} (A, \tilde{R}) &= \text{msg}((A_1, \tilde{R}_1), (A_2, \tilde{R}_2)) \\ B &= p(X_1, \ldots, X_n) \\ B_i &= p(t_1^i, \ldots, t_n^i) \\ (A, \tilde{R})\{X_1/t_1^i, \ldots, X_n/t_n^i\} &\simeq (A_i, \tilde{R}_i) \end{aligned}$$

An application of the second part of intra-construction also replaces (E_1) and (E_2) by (F). Compared with folding, not only must the definition clauses be unit clauses, but their argument places must be terms that undo the generalization when resolving (F) with (D_1) and (D_2). If we are introducing a new definition, such constraints might not be relevant. If, however, we wish to recognize an already-inferred concept or we are interested in increasing the efficiency of execution of our inferred program, it might be important to be able to relax such limitations.

2 A common extension of folding and intra-construction

As an illustration showing the need for increasing the efficiency of an inferred program, consider the *string-matching problem*. This problem consists in finding all occurrences (or only one occurrence, if any) of a string called the *pattern* within another string called the *text*. Suppose that our pattern is the string aab. If we give the following two examples, both of which are a text having an occurrence of this pattern:

$$s([a,a,b]) \leftarrow \quad (1)$$
$$s([a,a,a,b]) \leftarrow$$

we can apply absorption, deriving:

$$s([a|X]) \leftarrow s(X) \quad (2)$$

A third example:

$$s([b,a,a,b]) \leftarrow$$

allows us to apply absorption again, inferring:

$$s([b|X]) \leftarrow s(X) \quad (3)$$

Clauses (1), (2), and (3) can be regarded as representing a finite-state automaton accepting the language $(a+b)^*aab$. However, these clauses are a nondeterministic program, requiring a quadratic time in the worst case to find an occurrence of the pattern. Note that this program cannot be converted to a deterministic version with intra-construction. A reason is that we would need to be able to apply intra-construction without inventing a new predicate at every application. (Such already-existing predicates represent edges going backwards in the corresponding automaton.) Allowing the use of already-existing definitions is accompanied by a possible violation of the conditions for intra-construction (i.e. the B_i's may not have the required form).

This limitation suggests using the ordinary folding [Pettorossi and Proietti, 1998] rule as part of the operator repertoire of an ILP system. In fact, [Pettorossi *et al.*, 1996] show how to determinize a program similar to the one we just inferred with absorption (given as specification by the user), where the folding rule plays a central role.

The ordinary folding rule [Pettorossi and Proietti, 1998], however, is not adequate for inductive logic programming: We observed that this rule requires the set of atoms that do not match the bodies of the definition (i.e. A and $\tilde{R}$) be the same in each clause to be folded (i.e. (E_1) and (E_2)).

Now we give another illustration. Suppose an inductive-inference system has inferred the following description of an arch, where the example *arch*$(([\,],$ *beam*$,[\,]))$ is not given until later:

$$\begin{aligned} &\mathit{arch}([X|Y],\mathit{beam},[X|Y])) \leftarrow \mathit{brick\_or\_block}(X),\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{column}(Y) \quad (4)\\ &\mathit{column}([\,]) \leftarrow\\ &\mathit{column}([X|Y]) \leftarrow \mathit{brick\_or\_block}(X),\ \mathit{column}(Y) \end{aligned}$$

If we now add the example:

$$\mathit{arch}(([\,],\mathit{beam},[\,])) \leftarrow \quad (5)$$

both folding and intra-construction are unable to derive the desired concept from (4) and (5):

$$\mathit{arch}((X,\mathit{beam},X)) \leftarrow \mathit{column}(X) \quad (6)$$

Being able to derive the concise version of *arch* (6) might be important if we have already the same concept under a different name:

$$\mathit{gateway}((X,\mathit{beam},X)) \leftarrow \mathit{column}(X)$$

We have recently extended the ordinary folding rule so as to overcome these limitations of folding [Rosenblueth, 2005]. Our extension can be viewed as a common generalization of ordinary folding and intra-construction:

$$\frac{B_1 \leftarrow \tilde{Q}_1 \quad (\mathrm{D}_1) \qquad A_1 \leftarrow \tilde{Q}_1\theta_1,\tilde{R}_1 \quad (\mathrm{E}_1)}{A_1 \leftarrow B_1\theta_1,\tilde{R}_1 \quad (\mathrm{F}_1)} \qquad \frac{B_2 \leftarrow \tilde{Q}_2 \quad (\mathrm{D}_2) \qquad A_2 \leftarrow \tilde{Q}_2\theta_2,\tilde{R}_2 \quad (\mathrm{E}_2)}{A_2 \leftarrow B_2\theta_2,\tilde{R}_2 \quad (\mathrm{F}_2)}$$

$$\frac{A_1 \leftarrow B_1\theta_1,\tilde{R}_1 \quad (\mathrm{F}_1) \qquad A_2 \leftarrow B_2\theta_2,\tilde{R}_2 \quad (\mathrm{F}_2)}{A \leftarrow B,\tilde{R} \quad (\mathrm{F})}$$

where

$$(A,\tilde{R}) = \mathrm{msg}((A_1,\tilde{R}_1),(A_2,\tilde{R}_2))$$
$$B = \mathrm{msg}(B_1\theta_1\tau_1, B_2\theta_2\tau_2)$$

Here, θ_i only acts on the variables occurring in $\tilde{Q}_i$, and τ_i only acts on the variables occurring in B_i and not occurring in $\tilde{Q}_i$ (i.e. the *head-only* variables in (D_i)). In addition, we preserve the syntactic conditions on θ_i developed for ordinary (multiple-clause) folding [Pettorossi and Proietti, 1998].

The main point of our folding rule is the computation of the substitutions τ_i in such a way that soundness is preserved. We distinguish two kinds of head-only variable instantiation: (a) *coverage*, taking a function symbol or variable from the substitutions ρ_i, where $(A_i,\tilde{R}_i) = \mathrm{msg}((A_1,\tilde{R}_1),(A_2,\tilde{R}_2))\rho_i$ $(i=1,2)$, and (b) *matching*, taking the function symbol from the corresponding position in the clause of the other single-clause folding.

The contributions of this poster over [Rosenblueth, 2005] are (1) the comparison of folding with intra-construction and (2) the development of inductively inferred example programs. We are currently investigating the possibility of embedding our folding rule in CIGOL.

References

[Pettorossi and Proietti, 1998] Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs. In Dov M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.

[Pettorossi *et al.*, 1996] Alberto Pettorossi, Maurizio Proietti, and Sophie Renault. Enhancing partial deduction via unfold/fold rules. In *Proc. 6th Int. Workshop on Logic Program Synthesis and Transformation*, pages 146–168, Stockholm, Sweden, 1996. Springer-Verlag. Lecture Notes in Computer Science No. 1207.

[Rosenblueth, 2005] David A. Rosenblueth. A multiple-clause folding rule using instantiation and generalization. *submitted to Fundamenta Informaticae (Special Issue on Program Transformation)*, 2005.