International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

Data Flow Coherence Constraints for Pruning the Search Space in ILP Tools*

SMARANDA MURESAN
Department of Computer Science, Columbia University, New York, USA
smara@cs.columbia.edu

TUDOR MURESAN, RODICA POTOLEA
Department of Computer Science, Technical University of Cluj-Napoca, Romania
{tmuresan, potolea}@cs.utcluj.ro

In this paper we present a new method that uses data-flow coherence constraints in definite logic program generation. We outline three main advantages of these constraints supported by our results: i) drastically pruning the search space (around 90%), ii) reducing the set of positive examples and reducing or even removing the need for the set of negative examples, and iii) allowing the induction of predicates that are difficult or even impossible to generate by other methods. Besides these constraints, the approach takes into consideration the program termination condition for recursive predicates. The paper outlines some theoretical issues and implementation aspects of our system for automatic logic program induction.

Keywords: Inductive logic programming, automatic program generation, data flow coherence constraints, implementation

1. Introduction

Inductive Logic Programming (ILP), which combines the positive characteristics of Logic Programming and Machine Learning, involves the induction of logic programs from examples and background knowledge [1 2 3 4 5 6]. ILP has a strong potential for being applied to real-world problems in various domains (e.g. biology, natural language processing) [7].

The theoretic and algorithmic support for building an automatic logic program generation system is given in [8 9], by introducing the Mode Directed Inverse Entailment (MDIE) method, a state-of-the-art ILP technique. The mode declarations for both the head and the body of the learned hypotheses (*modeh* and *modeb* respectively) bound the most specific clause, which is built based on a positive example. The most specific clause is used to lower bound the hypothesis search space, and thus the search is better constrained than those in which the sub-lattice is only upper bounded by the empty clause. Then an A*-like al-

*A version of this paper was presented at ICTAI 2001 with the title "Data Flow Coherence Criteria in ILP Tools"

gorithm is used for searching the subsumption lattice in order to find the clause with maximal compression.

Our experimental tool for logic program generation [10] is based on MDIE, but it introduces new heuristics for further pruning the search space. The most important feature of our approach is that it imposes several data flow coherence constraints (transfer coherence criteria) applicable to definite logic program generation. In this approach, each predicate establishes a data flow from the input to the output arguments. Those predicates that have only input arguments represent constraints on the data, which in pure logic programming lead finally to unification. For recursive predicates a further constraint is added: a termination condition similar to that in [1 5 11]. These constraints are used both in the generation of the most specific clause and in the hypotheses searching.

The benchmark tests show that the number of the hypotheses that satisfy the data-flow coherence constraints is very small. Only these hypotheses are considered for verifying positive examples. Furthermore, our initial experimental results support the hypothesis that these coherence constraints allow learning of nondeterministic programs and enable learning only from positive examples for a class of logic programs . Section 2 of this paper presents the theoretical issues: the definition of the immediate transfer operator and representative examples, the generation of the most specific clause that satisfies the data-flow coherence constraints and the generation of the hypotheses with maximal compression. Section 3 describes the implementation of the system, which is done in SICStus Prolog, together with the program sequences that are generated (by metaprogramming technique) for searching both the most specific clause and the hypotheses. Section 4 presents the experimental results. Our conclusions are drawn in Section 5.

2. Data-flow Coherence Constraints and Representative Examples

2.1. *General Formal Framework for Logic Program Generation*

The general framework for logic program generation is as follows. Let M_K(h) be the least Herbrand model[14] of the background knowledge K restricted to the predicate symbol h and E^+ be the set of positive examples (the set of ground atoms) using e as the predicate symbol. E^+ is a rule set with empty antecedent (premise) set. If $M_K(\mathrm{e}) = \phi$ then $M_E = M_{K \bigcup E}$ (e) $= E^+$. Let H be the set of hypotheses learned from the positive examples E^+ and the background knowledge K. Generally from $K \wedge H \vdash E^+$ results $E^+ \subseteq M_{K \bigcup E}$ (e). If negative examples, E^-, are considered, then $M_{K \bigcup E}$ (e) $\bigcap E^- = \phi$, because $(K \wedge H \nvdash E^-)$. H represents a new rule set, which replaces the old rule set E^+, by increasing the least Herbrand model, which is bounded by the

negative examples. In the Mode Directed Inverse Entailment (MDIE) method, H is obtained based only on K-proving. The whole theoretical and algorithmic support for its construction is given in [8] : starting with the knowledge base, K, and a positive example $e \in E^+$ the most specific clause, $\perp_e$, is built, such that $K \wedge \perp_e \vdash e$. Then a refinement operator and an A* - like algorithm are used for finding the hypothesis (clause) with maximal compression. Finally a cover set algorithm is used for learning the whole set of hypotheses, H.

2.2. *Data-flow Coherence Criteria for Pruning the Search Space*

MDIE method has the advantage of both lower and upper bounding the hypotheses search space. In our approach we further prune the search space by introducing four data-flow coherence criteria. These criteria can sometimes substitute the set of negative examples, in their function of bounding the increase of the least Herbrand model of the hypotheses set. In our model, any predicate is seen as realizing a data flow from the input to the output arguments.

The immediate transfer operator we introduced in [10], T, specifies all terms that can be obtained in one step at the output of the predicates, based on the terms given as inputs. If T^i is the ith-power of T then T^0 represents the set of terms given by the inputs of the positive example (ground atom) $e \in E^+$. The successive powers of T generate all ground terms obtainable based on the input terms of the positive example, e, and background knowledge K. Let d be the minimum value such that T^d contains all ground output terms of the positive example e, and A_e the ground atoms set for which all atom terms are in T^d. The most specific hypothesis for $e \in E^+$ and K is defined as:

$$\perp_e = e \leftarrow b_1, b_2 ..., b_n$$

where $b_i \in A_e$ and all literals b_i satisfy the four data-flow coherence criteria defined below:

• **The First criterion** forces each output argument of the hypothesis head to be obtained either from an input argument of the head or from an output argument of a body literal, or to be a *bottom element* of the data type defined by *dftype* (e.g. the bottom element for list type is [], for tree type is nil and for natural numbers type is 0; see Section 3.1 and Appendix A)

• **The Second criterion** requires that each input of a body literal be obtained either from an input of the hypothesis head or from an output of a preceeding literal in the body.

• **The Third criterion** requires that each output of a body literal be linked either to an output of the hypothesis head or to an input of a following literal in the body.

• **The Fourth criterion** forbids that the inputs of the hypothesis head to be linked to the outputs of the body literals. Furthermore this criterion requires that all output variables of the body literals be pairwise distinct.

Any input of a literal in the body of $\perp_e$ is given by an output of a precedent

literal (the flow of data in the hypothesis body is from left to right, except for the literal outputs connected to head outputs). It should be noted that:

- not all atoms verify the transfer coherence criteria, i.e. $\exists a\,(a \in A_e \wedge a \notin \bot_e)$
- not all terms $t \in T^d$ satisfy the criteria .

In [10] we introduced the following theorem:

Theorem 1. *$K \wedge \bot_e \vdash e$ if T^d exists and d is a finite number. (e is provable from the background knowledge K and the most specific hypothesis.)*

We denote by $t \in \bot_e$ the terms that satisfy the transfer coherence criteria and by h the generalization of $\bot_e$ with substitution θ defined as :

$$\theta = \{v_i/t \,|\, t \in \bot_e\}$$

The fourth previous transfer coherence criterion for $\bot_e$ is changed for h into a new one, which states that the generalization variables v_i assigned to the hypothesis head inputs and to the body literals outputs must be different. It is always true that $h\theta \subseteq \bot_e$ and from the above theorem results that:

$$K \wedge h \vdash e$$

In general, from a learned hypothesis h, a subset E_h of positive examples is K-provable, i.e.:

$$K \wedge h \vdash E_h$$

where $E_h \subseteq E^+$

If H_e is the set of all h coherent hypotheses (Fig. 1), which generalize $\bot_e$, the one with maximum compression is chosen:

$$|E_h^*| = \max_{h \in H_e} |E_h|$$

If two hypotheses h and h' have the same maximum compression, the most specific one is chosen, i.e.:

If

$$|E_h| = |E_{h'}| = |E_h^*|$$

the final hypothesis h is chosen so that it minimally increases the declarative semantics, thus preventing overgeneralization, i.e. $M_{K \cup h}(e) \subset M_{K \cup h'}(e)$. Sometimes this can substitute for the lack of negative examples, thus our approach can be seen as a method of learning only from positive examples.

2.3. *Representative Examples. Partial Ordering Relation*

Besides the four data-flow criteria described above for pruning the search space, our method introduces a termination condition that is based on the partial ordering relation among the examples. Adapting a theorem from structural induction, we say that a rule set K over the Herbrand base B_K terminates iff there exists a well-founded relation $\succ$ on B_K such that for all rules $y \leftarrow X \in ground(K)$ and for all $x \in X$ we have that $y \succ x$. The recursive predicates impose that the relation $\succ$ holds for terms appearing as predicate arguments. The relation $\succ$ is a semantic and not a syntactic one. It is given by the descendant declarations (see Section 3.1 and Appendix A) and induces

a partial ordering over the set of examples E^{+} [5] [11] [12] (see Section 3.2).

We say that a set E_0 *is representative* for a problem if it is $K \wedge H$ - provable for any rule set H which models that problem. For any rule sets H and H' which model the problem, the following equivalence rule holds:

$$\frac{(K \wedge H \vdash E_0) \wedge (K \wedge H' \vdash E_0)}{\forall E^{+}\,[K \wedge H \vdash E^{+} \Leftrightarrow K \wedge H' \vdash E^{+}]} \tag{1}$$

Always $\exists\, e_0 \in E_0\, \forall e \in E^{+}\,[e_0 \not\succ e]$ such that all finite decreasing sequences of elements are ending in $e_0(e \succ ... \succ e_1 \succ e_0)$. We call e_o the *bottom elements* of the set E_0. Similarly we define the bottom elements of data types used as predicate arguments (defined by *dftype*). For e_0 results:

$\perp_{e_0} = e_0 \leftarrow b_1, b_2, ..., b_n$
or
$\perp_{e_0} = e_0 \leftarrow$

where b_i literals have the predicate symbol different from that of e_0 ($e_0 \succ b_i$ given by descendant) .

The termination condition on the hypothesis set H and the clauses from the previous equation impose that the cover set algorithm [8] must first extract e_0 from E^{+}. If the examples E^{+} are topologically sorted, the process continues as in Fig. 1. Thus, the partial ordering relation $\succ$ becomes mandatory for inducing the hypothesis set and for avoiding the inclusion in A_e of those atoms, which can generate infinite loops.

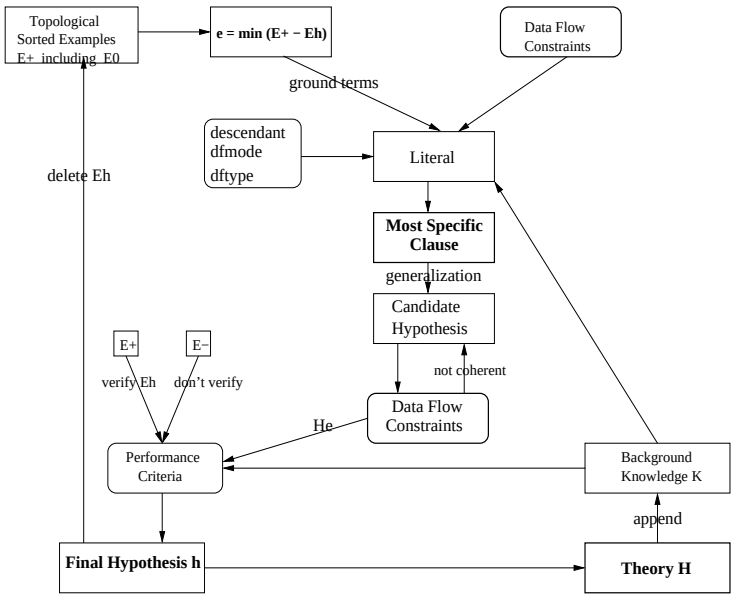

Fig. 1. Most Specific Clause and Hypotheses Generation + Cover Set Algorithm

3. Implementation issues

3.1. *Mode Declaration and Input Files*

In our present version of the system, the ground term encoding results in a representation that is based on the total sharing of the substructures [10]. This representation is able to provide the argument templates, so that the user has only to introduce the data flow and the type of arguments as well as the calling graph of the predicates (*descendant*). For example, for generating the predicate that generates all permutations, the mode declarations are:

```
descendant(perm,[delete,perm])
dfmode(perm(+list,-list))
dfmode(delete(+integer,-list,+list))
dftype(list, [])
dftype(list, [integer|list])
```

The clauses of both the *delete* predicate and the *list* type are given in the input file as background knowledge, K, represented as a Prolog program. The positive examples are given as Prolog ground atoms.

3.2. *Terms and Atoms Ordering Relation*

The partial ordering of the representative examples (used by the cover set algorithm) might be based on: 1) the number of distinct simple terms, or 2) the total number of simple terms, or 3) a user-specified order. In the first case it is given by the subset inclusion relation on the term simple component sets (A term T is simple if its arity is 0).

$$T_1 \succ T_2 \leftarrow T_1 \succeq T_2\,,\, not(T_2 \succeq T_1)$$
$$T_1 \succeq T_2 \leftarrow for\_all\, simple(T_2,\, Simple)$$
$$is\_simple(T_1,\, Simple)$$

Two ground atoms satisfy the order relation if at least one of the input arguments respects the order relation.

$$A_1 \succ A_2 \leftarrow input\_arg(N,\, A_1,\, I_1)\,,$$
$$input\_arg(N,\, A_2,\, I_2)\,,$$
$$I_1 \succ I_2$$

In the second case (2), the total number of simple terms is determined during the ground atom encoding.

3.3. *Most specific clause $\perp_e$ generation*

For generating $\perp_e$, the metaprogramming technique is used. This technique allows the generation of a Prolog program, whose execution generates, by means of backtracking, all the literals of the most specific ground atoms A_e . For a *descendant* predicate, which has as mode declaration: *dfmode(p(+type1, +type2, -type3)),* the following program is generated:

```
P = value(type1, T1, in, Id1),
    value(type2, T2, in, Id2),
    solve(p(T1, T2, T3), Hdepth_bound),
    ordering_test,
    mem_type(type3, T3, in, Id3),
    mem_literal(p(Id1, Id2, Id3)) .
```

where,

• *value* is used for assigning to each ground term T of type t, a unique index I_d

• *solve* is a depth-bound metainterpreter with backtracking

• *mem_ type* is used for storing a new ground term T of type t having the index I_d

• *mem_ literal* is used for storing the ground atoms of the most specific ground atom set A_e

• *ordering_ test* tests the order relation between the head of e and the ground atoms of A_e

By means of the following program sequence: *call(P), fail.*, all literals A_e are generated by asserting them in the Prolog database. After the values of the output arguments of the current positive example are obtained, all the literals and terms that do not satisfy the data-flow coherence criteria are retracted, thus obtaining $\perp_e$ (Fig. 1).

Let us consider the *perm* predicate specified in Section 3.1. For the positive example:

e_0 `= perm([],[])`

the most specific clause is:

$\perp_e$ `=perm(+t1,-t1)`

where t1=[ ].
For the positive example $e \succ e_0$:

```
e = perm([1],[1])
```

the most specific clause is:

```
⊥e = perm(+t3,-t3):- +t3=[-t2|-t1],
                     perm(+t1,-t1),
                     delete(+t2,-t3,+t1).
```

where t3=[1], t2=1 and t1=[ ].
It can be observed that the immediate transfer operator generates from t3 (the input term of the second positive example) the set T^d = {t3, t2, t1} and that the atoms of the most specific clause satisfy the data-flow coherence criteria.

3.4. *Hypothesis Generation*

A metaprogramming technique, similar to that presented above, is used for generating the hypotheses that generalize $\perp_e$. The P program is used in an algorithm that replaces the technique of the A*-like algorithm [8] with a depth search approach constrained by the four data-flow coherence criteria, for generating all partial hypotheses obtained by adding a new literal (Fig. 1). The partial hypotheses that have already become incoherent are not generated. Each generated partial hypothesis is verified only if it satisfies the transfer coherence criteria. If it does, it is stored as a final hypothesis, otherwise it is stored as candidate and the process of adding new literals is continued.

For a descendant predicate which has as mode declaration: *dfmode(p(-type1, +type2, -type3))* the following program is generated:

```
P = theta(-1,(V1, Atr_arg1)/Id1),
    verify_theta(J, (V1, Type_arg1 )/Id1),
    theta(J,(V2, Atr_arg2)/Id2),
    verify_theta(J,(V2, Type_arg2)/Id2),
    theta(-2,(V3, Atr_arg3)/Id3),
    verify_theta(J,(V3, Type_arg3)/Id3),
    new(Jn),  copy_theta(J, Jn),
    mem_hypothesis(hyp(Jn)),
(coherent → mem_final(h(Jn)); mem_candidate(h(Jn, L))).
```

where:

• *theta (J, _)* is the substitution set θ for the current hypothesis J that has associated the attribute of the predicate argument. The negative values for J denote the output arguments, for which θ contains also a new generalization variable. For the clause head, both input and output arguments have J negative, and thus they introduce a new generalization variable.

• *verify_ theta (J, _)* verifies immediately the incoherence, based on the attribute and the type of arguments. The attribute takes into consideration only the previous arguments that shared the variable, while the type refers only to the current argument. The hypotheses that have already become incoherent are rejected (*early pruned*), admitting only those that can become coherent by adding new literals.

• *copy_ theta (J, Jn)* copies the substitution of the current hypothesis J for the new generated partial hypothesis *Jn*.

• *mem_ hypothesis* stores the new hypothesis along with all the necessary information (not given here)

• *coherent* verifies the total coherence, that means that all the outputs from the hypothesis head and all the inputs and outputs from the hypothesis body must share a variable. The sharing compatibility was checked by *verify_ theta*. The final hypotheses that do not verify the total coherence are rejected.

• *mem_ candidate* stores the partial hypothesis *Jn* along with the L index of the new literal.

• *mem_ final* stores the hypotheses H_e that verify the transfer coherence criteria.

Using the following program sequence: *call(P), fail.*, all partial hypotheses obtained by expanding a candidate hypothesis (i.e. obtained by adding a new literal L) are generated.

In the end, the final hypotheses are tested on all positive (and when exist on negative) examples and the hypothesis with the maximum compression is chosen. At the same compression, the most specific hypothesis is taken.

In the above exemplified predicate (*perm*), for the positive examples e_0 and e results the following hypotheses that satisfy the data flow coherence criteria:

```
perm(+[],-[]).

perm(+V1,-V2) :- +V1=[-V3|-V4], delete(+V3,-V2,+V4). (1)
perm(+V1,-V2) :- +V1=[-V3|-V4], perm(+V4,-V6), delete(+V3,-V2,+V6). (2)
```

Using the depth first search approach and data-flow criteria the correct (in this case the longest (2)) hypothesis for the example e is chosen from the two final solutions based on maximal compression (E^+ pruning, see Table 1). This would not have been possible using an A*-like algorithm, which stops at the first found final solution (in this case the shortest (1)). Thus the final hypotheses set is:

```
perm([], []).
perm([A|B], C) :- perm(B, D), delete(A, C, D).
```

4. Experimental Results and Discussions

The system was tested on a set of test benchmarks for list and tree processing presented in Table 1 and Appendix A. The statistics show that 81.9% - 99.8% of the partial hypotheses are pruned as data flow incoherent, and that 56.2% - 96.8% are early pruned by *verify_ theta* predicate described in Section 3.4. Thus the number of data flow coherent hypotheses is very small, and only these hypotheses are tested on the positive and negative examples. Analyzing Table 1, it can be seen that a class of predicates does not need negative examples in the generation process, their induction being made based only on positive examples.

Table 1. Statistics of data flow coherence constraints effect on a set of test benchmarks

Predicate	K	$\|E^+\|$	$\|E^-\|$	DF pruned(%)		Hyp DF	Pruned by		$\|H\|$	Time
				early	final	coherent	E^-	E^+		(sec)
append(+, +, -)		5	0	76.7	88.3	5	0	3	2	0.03
append(-, -, +)		5	0	72.7	94.3	5	0	3	2	0.05
last(-, +)		2	1	56.2	87.5	2	0	0	2	0.01
reverse(+, -)	append_el(+, +, -)	3	1	63.6	81.9	4	1	1	2	0.02
delete(-, +, -)		4	0	78.6	97.1	2	0	0	2	0.05
delete(+, -, +)		4	0	80.0	95.6	2	0	0	2	0.03
bal_merge(+, +, -)		5	0	78.6	89.2	6	0	4	2	0.03
bal_split(+, -, -)		3	1	74.1	94.8	6	1	3	2	0.05
ord_merge(+, +, -)	=<(+, +)	9	2	78.3	96.2	90	51	35	4	2.01
partition(+, +, -, -)	=<(+, +)	6	6	84.7	98.6	24	15	6	3	1.38
merge_sort(+, -) weak typed	bal_split(+, -, -) ord_merge(+, +, -)	4	3	89.3	99.4	31	1	27	3	1.89
qsort(+, -) weak typed (App. A2)	partition(+, +, -, -) append(+, +, -)	3	1	96.8	99.8	79	1	76	2	9.14
qsort(+, -) strong typed (App. A2')	partition(+, +, -, -) append(+, +, -)	3	1	86.5	97.2	4	1	1	2	0.08
qsort(+, -) strong typed (App. A2")	partition(+, +, -, -) append(+, +, -)	3	1	88.8	96.5	6	1	3	2	0.10
perm(+, -)	delete(-, +, -)	4	0	79.6	94.4	3	0	1	2	0.05
		3	1	75.5	89.8	5	1	2	2	0.03
	delete(+, -, +)	4	0	65.0	85.0	3	0	1	2	0.04
		3	1	66.7	83.3	4	1	1	2	0.02
insert_tree(+, +, -)		6	0	68.5	99.3	5	0	2	3	0.58
search_tree(-, +)		6	0	65.6	90.6	3	0	0	3	0.04
insert_ord_tree(+, +, -)	=(+, +) <(+, +)	5	3	67.2	99.6	7	2	2	3	1.62
search_ord_tree(+, +, -)	=(+, +) <(+, +)	6	3	63.5	90.4	11	1	7	3	0.11
profile(+, -)	append(+, +, -)	3	0	71.8	92.3	6	0	4	2	0.07
profile_diff(+, -, +)		4	0	82.4	95.1	5	0	3	2	0.06
same_profile(+, +)	profile(+, -) =(+, +)	2	0	64.7	82.4	3	0	2	1	0.02
merge_tree(+, +, -)	profile(+, -) ord_merge(+, +, -)	2	0	65.6	84.3	5	0	4	1	0.03
DCG top down parser s(-, +, -)	np(-, +, -) vp(-, +, -)	1	0	73.9	97.8	1	0	0	1	0.06
DCG top down parser s(+, -, +)	np(+, -, +) vp(+, -, +)	1	0	64.3	92.9	1	0	0	1	0.03

All these pruning techniques reduce drastically the size of the set of positive and negative examples on which the induction process is performed (e.g. for *qsort/2* only three positive and one negative example are needed). The number of representative examples selected by the cover set algorithm is equal with the size of the final set of hypotheses (*Theory H* in Fig. 1).

All predicates presented in Table 1 are recursive, but the last four. For these the induction is based only on one or two positive examples and no negative examples. The last two examples represent the induction of a DCG top down reversible parser given in [13]. The set H of hypotheses induced for predicate *s/3* (as well as for *append/3* and *delete/3*) is the same regardless of the argument data flow switching. This feature, along with the ability of learning recursive rules based only on positive examples makes the system a suited tool for natural language processing applications.

The system allows the induction of a predicate either starting with a background knowledge, K, (see Appendix A3, A7), or by performing a bottom up induction of all predicates specified in the *descendant* declaration, thus inducing also the background knowledge from mode, type declarations and examples (see Appendix A1, A1', A2, A2', A2", A6).

The generation of the final set of hypotheses H is done incrementally by adding one clause at each iteration step. Each partial set of hypotheses is tested on all positive examples (and when exist on all negative examples) using a depth-bound metainterpreter with backtracking (Fig. 1). Thus the final set of hypotheses H, can be seen as a non-deterministic definite logic program, which verifies all positive examples and rejects all negative examples. The non-deterministic programs are obtained only if there exist positive examples which for the same values of the input arguments have different values of the output arguments (see Appendix A1)

If more than one proof tree exists for a representative example e, a cut ($!$) operator is added (see Appendix A1', A3, A6) and the definite logic program becomes deterministic. It must be pointed out that for *merge_sort/2* predicate presented in Appendix A3, both cut ($!$) operators were added when the third clause was generated in the set of hypotheses. At this point the representative examples used for generating the first two clauses of predicate *merge_sort*, have at the same time more than one proof tree. Thus, in order to generate a deterministic program the cut operator was added to these two clauses.

The running times presented in Table 1 are obtained on a 1 GHz Pentium III. For *qsort* predicate, the time can be drastically improved if we use strong typing by introducing the sorted list type (*slist*) (see Table 1 and Appendix A2' and A2"). The solutions in Appendix A2' and A2" are different, but both are correct and verify the equivalence rule for the representative examples (see Section 2.3). In the case of *qsort* generation with weak typing, presented in Appendix A2, both these solutions are obtained but the 'performance criteria' (Fig. 1) choose the most efficient one (i.e the solution for which the metainterpreter performs a minimum number of resolution steps).

In this version of the system, we assume that the set of positive examples contains the representative examples (Fig. 1). An important feature of the

system is that it allows weak and strong data flow coherence criteria that can be specified by the user. On one hand, the weak data flow coherence allows the presence of free variables, thus increasing the expressiveness of the generated program (Appendix A5). On the other hand, the strong data flow coherence links an output variable to exactly one input variable, thus increasing the efficiency (Appendix A3).

The results presented in this section constitute the grounds for experimentally validating a method that uses data-flow coherence constraints for pruning the search space and enables learning a class of logic programs only from positive examples. However, the negative examples are used when they are absolutely necessary, especially for distinguishing a problem from others (e.g. *search_ord_tree* from *search_tree*, see Appendix A4, A5 and Table 1).

5. Conclusions

We presented a method for automatic program generation, implemented in an experimental system based on MDIE method used in [8]. The original theoretical and implementation contributions of our work are:

Theoretic:

- use of data flow coherence constraints
- generation of the most specific clause using the immediate transfer operator
- generation of the hypotheses set using data flow coherence criteria
- use of the topologically sorted representative examples in the cover set algorithm
- use of strong typing to increase the time efficiency

Implementation:

- use of metaprogramming technique
- use of a bound metainterpreter with backtracking

The effect of these contributions is: i) drastically prune the search space (around 90%), ii) reduce the set of positive examples and reduce or even remove the need for the set of negative examples, and iii) allow the induction of predicates that are difficult or even impossible to generate by other methods.

Only the coherent hypotheses are verified on the positive examples in order to determine the one with the maximal compression. For a large class of logic programs the learning can be done efficiently based only on positive examples. The benchmark set on which the system was tested emphasized the advantages of the Inverse Entailment method, which can benefit from all the prover characteristics (e.g. the nondeterministic search by backtracking).

Furthermore the system can be seen as the basis of a new Logic Programming style based only on declarations and examples that will generate the predicate in a bottom up manner, based both on the user specified calling graph and on the predicate arguments data flow.

The obtained results encourage us to pursue a new algorithmic approach based on data-flow coherence constraints and specialized for Natural Language

Processing, which we are currently exploring.

Acknowledgments

The authors would like to thank Judith Klavans and Chris Okasaki for insightful comments and feedback on drafts of this work.

References

[1] F. Bergadano and D. Gunetti, *An Interactive System to learn Functional Logic Programs*, Proc Thirteenth International Joint Conference on Artificial Intelligence, (1993) 1044–1049.

[2] T.M Mitchell, *Machine Learning*, McGraw-Hill (1997) 274–301.

[3] S. Muggleton, *Inductive Logic Programming*, New Generation Computing (1991) 295–318.

[4] J. Quinlan, *Learning Logical Definitions from Relations*, Machine Learning (1990) 239–266.

[5] J. Quinlan, *Learning First-Order Definitions of Functions*, Journal of Artificial Intelligence Research **5** (1996) 139–161.

[6] J.M Zelle and R.J Mooney, *Combining FOIL and EBG to Speed-up Logic Programs*, Proc. of the Thirteenth International Joint Conference on Artificial Intelligence, Chambery, France (1993) 1106–1111.

[7] R.J Mooney and M.E Califf, *Induction of First-Order Decision Lists: Results on learning the Past Tense of English Verbs*, Journal of Artificial Intelligence Research **3** (1995) 1–24.

[8] S. Muggleton, *Inverse Entailment and Progol*, New Generation Computing (1995) 245–286.

[9] S. Muggleton, *Completing Inverse Entailment*, in Page, C.D (ed), Proc of the Eight International Workshop on Inductive Logic Programming (ILP-98), Springer-Verlag, Berlin (1998) 245–249.

[10] S. Muresan, T. Muresan and R. Potolea, *An Automatic Logic Program Generation Kernel (ALPGK)*, Automation Computers Applied Mathematics, Scientific Journal, **8-1**, Technical University of Cluj Napoca (1999) 28–43.

[11] M. Proietti and A. Pettorossi, *Transforming Inductive Definitions*, Proc. of the 1999 International Conference on Logic Programming, Las Cruces, New Mexico (1999).

[12] K. Apt, *From Logic Programming to Prolog*, Prentice Hall Europe (1997).

[13] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press (1986).

[14] J.W. Lloyd, *Foundation of Logic Programming*, Springer-Verlag, Berlin (1987).

Appendix A Generated Logic Programs

A1. Bottom Up Induction of nondeterministic perm/2

```
START perm/2 induction
descendant(perm, [delete,perm]).
descendant(delete, [delete]).
dfmode(perm(+list,-list)).
dfmode(delete(-integer,+list,-list)).
dftype(list, []).
dftype(list, [integer|list]).

delete/3 induction % nondeterministic
delete(A, [A|B], B).
delete(A, [B|C], [B|D]) :-
        delete(A, C, D).

Final set H of hypoteses for perm/2
 % |E-|=1
perm([], []).  % nondeterministic
perm(A, [B|C]) :-
        delete(B, A, D),
        perm(D, C).
```

A1'. Bottom Up Induction of deterministic perm/2

```
START perm/2 induction
descendant(perm, [delete,perm]).
descendant(delete, [delete]).
dfmode(perm(+list,-list)).
dfmode(delete(-integer,+list,-list)).
dftype(list, []).
dftype(list, [integer|list]).

delete/3 induction % nondeterministic
delete(A, [A|B], B).
delete(A, [B|C], [B|D]) :-
        delete(A, C, D).

Final set H of hypoteses for perm/2
 % |E-|=0
perm(X, X) :-!.  % deterministic
perm(A, [B|C]) :-
        delete(B, A, D),
        perm(D, C).
```

A2. Bottom Up Induction of qsort/2 with weak typing

```
START qsort/2 induction
descendant(qsort,
  [partition,qsort,append]).
descendant(partition,
  [=<,partition]).
descendant(append, [append]).
dfmode(qsort(+list,-list)).
dfmode(partition(+integer,+list,
                 -list,-list)).
dfmode(append(+list,+list,-list)).
dfmode(+integer=< +integer).
dftype(list, []).
dftype(list, [integer|list]).

partition/4 induction
partition(_, [], [], []).
partition(A, [B|C], D, [B|E]) :-
        A=<B,
        partition(A, C, D, E).
partition(A, [B|C], [B|D], E) :-
        B=<A,
        partition(A, C, D, E).

append/3 induction
append([], A, A).
append([A|B], C, [A|D]) :-
        append(B, C, D).

Final set H of hypoteses for qsort/2
qsort([], []).
qsort([A|B], C) :-
        partition(A, B, D, E),
        qsort(E, F),
        qsort(D, G),
        append(G, [A|F], C).
```

A2'. Bottom Up Induction of qsort/2 with strong typing (1)

```
START qsort/2 induction
descendant(qsort,
  [partition,qsort,append]).
descendant(partition,
  [=<,partition]).
descendant(append, [append]).
 dfmode(qsort(+list,-slist)).
dfmode(partition(+integer,+list,
                 -list,-list)).
 dfmode(append(+slist,+slist,-slist)).
dfmode(+integer=< +integer).
dftype(list, []).
dftype(list, [integer|list]).
 dftype(slist, []).
dftype(slist, [integer|slist]).

partition/4 induction
partition(_, [], [], []).
partition(A, [B|C], D, [B|E]) :-
        A=<B,
        partition(A, C, D, E).
```

```
partition(A, [B|C], [B|D], E) :-
        B=<A,
        partition(A, C, D, E).

append/3 induction
append([], A, A).
append([A|B], C, [A|D]) :-
        append(B, C, D).

Final set H of hypoteses for qsort/2
qsort([], []).
qsort([A|B], C) :-
        partition(A, B, D, E),
        qsort(E, F),
        qsort(D, G),
        append(G, [A|F], C).
```

A2". Bottom Up Induction of qsort/2 with strong typing (2)

```
START qsort/2 induction
descendant(qsort,
  [partition,qsort,append]).
descendant(partition, [=<,partition]).
descendant(append, [append]).
dfmode(qsort(+list,-slist)).
dfmode(partition(+integer,+slist,
-slist,-slist)).
dfmode(append(+slist,+slist,-slist)).
dfmode(+integer=< +integer).
dftype(list, []).
dftype(list, [integer|list]).
dftype(slist, []).
dftype(slist, [integer|slist]).

partition/4 induction
partition(_, [], [], []).
partition(A, [B|C], D, [B|E]) :-
        A=<B,
        partition(A, C, D, E).
partition(A, [B|C], [B|D], E) :-
        B=<A,
        partition(A, C, D, E).

append/3 induction
append([], A, A).
append([A|B], C, [A|D]) :-
        append(B, C, D).

Final set H of hypoteses for qsort/2
qsort([], []).
qsort([A|B], C) :-
        qsort(B, D),
        partition(A, D, E, F),
        append(E, [A|F], C).
```

A3. merge_sort/2 Induction with Background Knowledge

```
START merge_sort/2 induction
descendant(merge_sort,
  [bal_split,merge_sort,ord_merge]).
dfmode(bal_split(+list,-list,-list)).
dfmode(merge_sort(+list,-list)).
dfmode(ord_merge(+list,+list,-list)).
dftype(list, []).
dftype(list, [integer|list]).

Background knowledge:
bal_split([], [], []).
bal_split([A|B], [A|C], D) :-
        bal_split(B, D, C).

ord_merge(A, [], A).
ord_merge([], A, A).
ord_merge([A|B], [C|D], [C|E]) :-
        C=<A,
        ord_merge([A|B], D, E).
ord_merge([A|B], [C|D], [A|E]) :-
        A=<C,
        ord_merge([C|D], B, E).

Final set H of hypoteses for merge_sort/2
merge_sort([], []) :-!.
merge_sort([A], [A]) :-!.
merge_sort(A, B) :-
        bal_split(A, C, D),
        merge_sort(D, E),
        merge_sort(C, F),
        ord_merge(E, F, B).
```

A4. insert_ord_tree/3 Induction

```
START insert_ord_tree/3 induction
descendant(insert_ord_tree,
  [=,<,insert_ord_tree]).
dfmode(insert_ord_tree(+integer,
                       +tree,-tree)).
dfmode(+integer= +integer).
dfmode(+integer< +integer).
dftype(tree, nil).
dftype(tree, t(tree,integer,tree)).

Final set H of hypoteses for
insert_ord_tree/3

insert_ord_tree(A, nil, t(nil,A,nil)).
insert_ord_tree(A, t(B,C,D), t(B,C,E)) :-
        C<A,
        insert_ord_tree(A, D, E).
insert_ord_tree(A, t(B,C,D), t(E,C,D)) :-
        A<C,
        insert_ord_tree(A, B, E).
```

A5. search_ord_tree/3 Induction

```
START search_ord_tree/3 induction
descendant(search_ord_tree,
  [<,==,search_ord_tree]).
dfmode(search_ord_tree(+integer,
                   +tree,-name)).
dfmode(+integer< +integer).
dfmode(+integer== +integer).
dftype(tree, nil).
dftype(tree, t(tree,r(integer,name),
       tree)).

Final set H of hypoteses for
search_ord_tree/3
search_ord_tree(A, t(_,r(B,C),_), C) :-
        A==B.
search_ord_tree(A, t(_,r(B,_),C), D) :-
        B<A,
        search_ord_tree(A, C, D).
search_ord_tree(A, t(B,r(C,_),_), D) :-
        A<C,
        search_ord_tree(A, B, D).
```

A6. Bottom Up Induction of merge_tree/3

```
START merge_tree/3 induction
descendant(merge_tree,
  [profile,ord_merge]).
descendant(profile,
  [profile,append]).
descendant(append, [append]).
descendant(ord_merge,
  [=<,>,ord_merge]).
dfmode(merge_tree(+tree,+tree,-list)).
dfmode(profile(+tree,-list)).
dfmode(ord_merge(+list,+list,-list)).
dfmode(append(+list,+list,-list)).
dfmode(+integer=< +integer).
dfmode(+integer> +integer).
dftype(tree, nil).
dftype(tree, t(tree,integer,tree)).
dftype(list, []).
dftype(list, [integer|list]).

profile/2 induction

append/3 induction
append([], A, A).
append([A|B], C, [A|D]) :-
        append(B, C, D).

profile(nil, []).
profile(t(A,B,C), D) :-
        profile(C, E),
        profile(A, F),
        append(F, [B|E], D).

ord_merge/3 induction
ord_merge(A, [], A) :-!.
ord_merge([], A, A).
ord_merge([A|B], [C|D], [C|E]) :-
        C=<A,
        ord_merge([A|B], D, E).
ord_merge([A|B], [C|D], [A|E]) :-
        A=<C,
        ord_merge([C|D], B, E).

Final set H of hypoteses for
merge_tree/3
merge_tree(A, B, C) :-
        profile(B, D),
        profile(A, E),
        ord_merge(D, E, C).
```

A7. DCG Top Down Reversible Parser Induction

```
START s/3 induction
descendant(s, [np,vp]).
dfmode(s(-stree,+wlist,-wlist)).
dfmode(np(-nptree,+wlist,-wlist)).
dfmode(vp(-vptree,+wlist,-wlist)).
dftype(wlist, []).
dftype(wlist, [atom|wlist]).
dftype(stree, s(nptree,vptree)).

Background knowledge:
np(np(D,N))-->det(D),np2(N).
np2(np2(A,N))-->adj(A),np2(N).
np2(np2(N))-->n(N).
vp(vp(V,N))-->v(V),np(N).
vp(vp(V))-->v(V).
det(det(this))-->[this].
det(det(a))-->[a].
n(n(boy))-->[boy].
n(n(dog))-->[dog].
adj(adj(nice))-->[nice].
adj(adj(brown))-->[brown].
v(v(sees))-->[sees].

Final set H of hypoteses for s/3
s(s(A,B), C, D) :-
        np(A, C, E),
        vp(B, E, D).
```