Constraint Inductive Logic Programming

Michèle Sebag [(1,2)] and Céline Rouveirol [(2)]
(1) LMS, CNRS-URA 317, Ecole Polytechnique, 91128 Palaiseau
(2) LRI, CNRS-URA 410, Université Paris Sud, 91405 Orsay
FRANCE

Abstract. This paper is concerned with learning from positive and negative examples expressed in first-order logic with numerical constants. The presented approach is based on the cooperation of Inductive Logic Programming (ILP) and Constraint Logic Programming (CLP), and proceeds as follows:

• A discriminant induction problem is shown to be equivalent to a Constraint Satisfaction Problem (CSP): all constrained clauses covering positive examples and rejecting negative examples can be trivially derived from the solutions of this CSP.

• Solving this CSP then allows to build the G set of solutions in terms of Version Spaces; this resolution can be delegated to a constraint solver.

• This CSP provides a tractable computational characterization of G, which is sufficient to classify further examples and offers simple counting-based heuristics to resist noisy data.

In this hybrid ILP-CLP approach, CLP performs most of the search involved in inductive learning; the advantage is to benefit from the number-handling capabilities of CLP, without requirement for additional background theory.

1 Introduction

This paper is devoted to learning from positive and negative examples expressed in first-order logic. Many learners have been developed in the field of Inductive Logic Programming (ILP): see [18, 22, 3, 24, 2, 13, 19], among others.

However, Logic Programming (LP), and consequently ILP, does not allow for efficient handling of numbers: as emphasized by Saraswat [26], *all concepts and operations of interest in [their] underlying domain of computation must be explicitly encoded in the form of Herbrand terms and operations on them*, which proves to be rather inefficient when dealing with continuous constants.

To overcome this limitation, this paper investigates a mode of cooperation between ILP and Constraint Logic Programming (CLP). This cooperation relies on the transformation of negative examples into constraints [27, 28, 30]. The first step is to build a clause $\mathcal{C}$ that subsumes both positive and negative examples. Some previous work was devoted to exploring the search space defined by the substitutions of clause $\mathcal{C}$ [30]: discriminant induction amounts to finding substitutions compatible with the positive example and incompatible with the negative examples.

This paper extends previous work in the constraint logic programming frame, which subsumes the logic programming frame [10]. The space of *substitutions of a clause* is embedded into the space of *constraints on the variables in this clause*; working in the latter space allows for dealing with more powerful and compact specializations.

We show that a discriminant induction problem can be automatically transformed into a Constraint Satisfaction Problem (CSP). This CSP gives ways to achieve either lazy or classical learning:

Building this CSP corresponds to a lazy learning mode, which is sufficient to classify unseen examples: We show that a new incoming example E satisfies the target concept (i.e., is subsumed by the body of a clause solution) iff there exists a substitution of $\mathcal{C}$ associated to E that satisfies this CSP.

Solving this CSP corresponds to classical learning:
All discriminant constrained clauses in the considered CLP language, can be trivially derived from the solutions of this CSP; and these solutions can be obtained via some constraint solver [1, 4, 5], i.e., algorithms designed for efficiently solving large combinatorial problems on finite and continuous domains.
The search part of induction can thereby be delegated to powerful external tools. ILP thus benefits from their sophisticated number-handling abilities without the need for additional background knowledge.

This paper is organized as follows. Section 2 gives a short introduction to CLP. In section 3, we present a transformation mapping a discriminant induction problem into a CSP. The inverse mapping permits to build the set of all discriminant constrained clauses, from the solutions of this CSP. The complexity issue is then examined, for both the resolution of this CSP, and its use to classify new examples. Finally, section 4 discusses the proposed approach with respect to some related works.

2 Maximally Discriminant Constraint Clauses

The discovery of clauses including simple numerical relations, (e.g., *The device breaks if the left crack is larger than the right crack by at least 2 inches*) requires some additional background theory (for instance, the definition of predicate $>$) to be provided, for all ILP learners we are aware of. Our goal is to overcome this limitation, and in particular, to be able to specify the domain of instantiation of numerical variables, or the interrelations of a pair of numerical variables, without having to provide the learner with the definition of those relations. This section thus investigates the formalism of constraint logic programming, for it both subsumes logic programming [10] and handles clauses that would require an additional theory in order to be discovered in ILP.

Of course, our aim is not to present here an exhaustive state of the art in CLP, but rather to define the basic CLP terminology and its underlying notions with respect to the classical LP and ILP terminology [14, 17].

2.1 CLP as an extension of Logic Programming

After [10], CLP languages extend the LP languages (traditionally, the language of definite clauses for PROLOG [14]), in the following sense.
The domain of computation for logic programs is the Herbrand universe; computations of logic programs can be brought back to solving equations in the Herbrand universe [12], which has the immediate consequence that every semantic object has to be coded into a Herbrand term.

The main motivation underlying CLP is to extend the domain of computation of programs to a structure in which computation can be expressed as *constraint solving*. This requires the introduction of an *algebraic* semantics. This scheme thus subsumes the LP scheme as equation solving is a special case of constraint solving.
As an example [10], consider the computation domain given by a two sorted algebra $\mathcal{S} = \mathcal{H} \cup \mathcal{A}$, where $\mathcal{H}$ is the Herbrand universe, (with uninterpreted functors, for which only unification may apply) and $\mathcal{A}$ is the set of natural integers, with the usual interpretation, and with (interpreted) functors such as $=$, $\neq$, $+$, $<$, $\in$.

2.2 Notations and definitions

Let us give some basic notations and definitions for CLP. Let $\mathcal{L}$ be a definite clause language without function symbols other than constants, $\mathcal{L} = \mathcal{L}_h \cup \mathcal{L}_c$, such that $\mathcal{L}_h$ defines the set of uninterpreted predicate symbols, whereas $\mathcal{L}_c$ defines the set of interpreted predicate symbols.

Definition 2.1 *A* constraint *is an atom built on a predicate symbol in $\mathcal{L}_c$. We will call in the remainder of the paper an* atom *(somehow unproperly, and as opposed to a constraint) a literal built on a predicate symbol in $\mathcal{L}_h$.*

Definition 2.2 *(after [31]). A* constrained clause *is a clause of the form:*

$$H :- B_1 \wedge \ldots \wedge B_m \wedge c_1 \wedge \ldots \wedge c_n$$

where H, $B_1, \ldots, B_m$ are atoms and $c_1, \ldots, c_n$ are constraints. In the following, we will refer to $c_1 \wedge \ldots \wedge c_n$ as to the constraint part *of the constrained clause, and to $B_1 \wedge \ldots \wedge B_m$ as to the* definite part *of the constrained clause.*

A constrained logic program *is a finite set of constrained clauses.*

A constraint goal *is a clause of the form: $:- B_1 \wedge \ldots \wedge B_m \wedge c_1 \wedge \ldots \wedge c_n$, where $B_1, \ldots, B_m$ are atoms and $c_1, \ldots, c_n$ are constraints.*

2.3 Operational Semantics of CLP language

The operational semantics of a CLP language can be defined either in terms of logical consequences or in an algebraic way [32] (see [10] for a detailed discussion).

Definition 2.3 *An* answer *to a CLP goal G is a conjunction of constraints $c_1 \wedge \ldots \wedge c_n$ such that*

$$P, \mathcal{T} \models (\forall)(c_1 \wedge \ldots \wedge c_n \rightarrow G), \text{ or}$$
$$P \models_{\mathcal{S}} (\forall)(c_1 \wedge \ldots \wedge c_n \rightarrow G),$$

where P is a constraint logic program, $\mathcal{S}$ is a structure, $\mathcal{T}$ is the theory axiomatizing $\mathcal{S}$ and $(\forall)F$ denotes the universal closure of F.

In the remainder of the paper, after [31], we use the only notation $\mathcal{D} \models$, which may be read both as the logical or algebraic version of logical entailment.

Definition 2.4 *A constraint c is* consistent *(or* satisfiable*) iff $\mathcal{D} \models (\exists)c$, which means that there exists at least one instantiation of variables of c in $\mathcal{D}$ such that c is true.*

More generally, given a conjunction of constraints σ and a constraint c, c is consistent with σ, *iff* $\mathcal{D} \models (\exists)(\sigma \wedge c)$

A (set of) constraint(s) σ is inconsistent *iff* $\mathcal{D} \models (\forall)(\neg\sigma))$.

Given a constraint c' and a conjunction of constraints σ, c' is entailed *by σ iff* $\mathcal{D} \models (\forall)(\sigma \rightarrow c')$

In LP, an answer to a query G with respect to a logic program P is a substitution σ (expressed as a set of equalities on variables of G) such that $G\sigma$ belongs to the least Herbrand model of P. An answer to a CLP program P is not anymore a substitution, but a set of consistent constraints such that all atoms in G have been resolved:

Definition 2.5 *[31] A* CLP resolvent *is denoted by (G, σ), where G denotes a conjunction of atoms and σ is a consistent conjunction of constraints.*

Definition 2.6 *The inference rule in CLP that generalizes resolution to CLP clauses can be modeled as follows. Given that:*

1. $C_s : H \leftarrow B_1 \wedge \ldots \wedge B_m \wedge c_1 \wedge \ldots \wedge c_n$, *where C_s is the selected clause of P*
2. $\mathcal{D} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A)$

Then, $(A \wedge G, \sigma)$ can be rewritten into $(B_1 \wedge \ldots \wedge B_m \wedge G, \sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A)$

The current CLP resolvent is $(A \wedge G, \sigma)$, A being the selected atom in the CLP resolvent.**(2)** denotes that the set of constraints σ, together with the constraints in C_s and the constraint stating the equality of the selected atom in the CLP resolvent and the head of the selected clause of P, are consistent. If **(1)** and **(2)** are true, a new CLP resolvent is obtained by replacing A in the current CLP resolvent, by the body of the selected rule (this parallels the classical resolution inference rule), and by adding the equality constraint stating that $A = H$. The derivation using such an inference rule succeeds when all atoms in the body of the resolvent have been deleted. The answer is therefore a consistent set of constraints such that atoms in the CLP goal are true, given the logic program P and the domain of computation of the constraints.

The usual resolution engine interacts with a constraint solver, that allows to decide for point **(2)**. In order to have a complete transition system, one should have a complete constraint solver, that is, a constraint solver that returns *true* if a constraint is consistent and *false* otherwise.

We now briefly present the type of constraints used in the remainder of this paper.

2.4 CLP languages

There is a huge number of CLP languages: without pretending to exhaustivity, see PROLOG III [4], CLP($\mathcal{R}$) [1], CHIP [5] (an extensive survey can be found in [11]). These languages tackle different domains of computation, e.g. boolean algebras, finite domains, rationals, reals, order-sorted feature trees. For some types of constraints (but not all), there exists complete and *efficient* constraint solvers that allow the decision about consistency and entailment of constraints.

In the present work, we restrict ourselves to the following types of constraints defined on linear (real or integer-valued), nominal and tree-structured variables:

- *domain* constraints: $(X \in [a,b])$, if X is a linear variable; $(X \leq a)$ if X is a tree-structured variable; $(X \in \{a_1, a_2, ..a_I\})$ or $(X \notin \{a_1, a_2, ..a_K\})$ if X is a nominal variable.
- *equality and inequality* binary constraints: $(X = Y)$ or $(X \neq Y)$.
- *binary arithmetic linear* constraints on linear variables: $(X - Y \geq a)$, $(X - Y \leq b)$.

Note that any substitution can be viewed as a conjunction of domain and equality constraints: a grounding is a special case of domain constraint, and links between variables can be expressed via equality constraints.

2.5 Constraint Programming

Definition 2.7 *A* constrained variable X *is a variable associated to a (possibly infinite) set of values, referred to as the* domain *of the variable and denoted* $dom(X)$.

A constraint satisfaction problem *is defined as a set of constrained variables and a set of constraints on these variables.*

Solving a CSP is finding one (or all) assignation(s) in their domain for each of the constrained variables such that all constraints are satisfied.

Solving a CSP is a highly combinatorial task. In order to have efficient search strategies solve CSPs, techniques such as *forward checking* have been developed to reduce the CSP into a hopefully minimal equivalent CSP. The idea underlying all such techniques is to dynamically reduce the domains of constrained variables by removing inconsistent values as soon as possible.

This is done by actively making use of the constraints. Just to give a flavor of how propagation operates, let us assume that variable X ranges in $[3,7]$ and variable Y ranges in $[2,5]$, and that a binary constraint additionally sets $X + Y < 8$. Then, the admissible domain for X may be reduced (e.g. $[3,6]$); an instantiation of X may have effects on admissible values for Y, and so on.

3 Mapping from ILP to CLP

This section develops a framework for learning all maximally general discriminant constrained clauses, i.e., in terms of Version Spaces, for building the G set in the above CLP language (section 2.4). We first transform such induction problem into a Constraint Satisfaction Problem (CSP). Then, it is shown that all constrained clauses in G can be derived from the solutions of CSP. Besides, this CSP can be used to classify unseen examples with a tractable complexity.

3.1 The search space

Our goal is to build all maximally discriminant constrained clauses that cover a positive example Ex and reject negative examples $Ce_1, ..Ce_n$. The limitation to a unique positive example can easily be overcome (4.4). The choice of maximally discriminant generalization is discussed in 4.1.

3.1.1 Language of instances and generalizations

The language of instances is that of (not necessarily ground) clauses without function symbols other than constants.The head of an example is a positive or negative literal built on the target predicate tc, and the body of an example is a conjunction of positive literals.

Our approach will be illustrated on a toy data set: Positive example Ex states that some 20-year old a, that is used by Tom for 5 years, the speed of which is 120 miles per hour, fails when used by Tom. Negative example Ce_1 states that any 10-year old stuff X, that is used by both Y since 10 years, and Z since recently, does not fail when used by Y. Last, negative example Ce_2 states that some b, that is used by Ann for an unknown period T, the speed of which is 0, does not fail when used by Ann.

Ex	$fail(a, tom) \leftarrow\ uses(a, tom, 5),\ speed(a, 120),\ age(a, 20)$
Ce_1	$\neg fail(X, Y) \leftarrow\ uses(X, Y, 10),\ uses(X, Z, 0),\ age(X, 10)$
Ce_2	$\neg fail(b, ann) \leftarrow\ uses(b, ann, T),\ speed(b, 0)$

The language of generalizations is that of constrained clauses (section 2.2).

$$g : tc(\vec{X}) \leftarrow L_h(\vec{X}, \vec{Y}), L_c(\vec{X}, \vec{Y}).$$

where the head of the constrained clause is a positive literal built on the target predicate, its definite part $L_h(\vec{X}, \vec{Y})$ involves the variables in the head of g, noted $\vec{X}$, and other variables, noted $\vec{Y}$, and its constraint part $L_c(\vec{X}, \vec{Y})$ is a conjunction of constraints set on variables in $\vec{X}$ and $\vec{Y}$.
In our example, the following constrained clause is a maximally discriminant generalization of Ex against Ce_1 and Ce_2:

$$fail(X, Y) \leftarrow uses(X, Y, Z), age(X, U), Z < U - 10$$

3.1.2 Generality order

We shall use, as in [30], the generality order given by θ-subsumption. We define the subsumption of a clause by a constrained clause as follows:

Definition 3.1 *Let C be a constrained clause, $C : tc(\vec{X}) \leftarrow L_h(\vec{X}, \vec{Y}) \wedge L_c(\vec{X}, \vec{Y})$, and let E be a clause. Then, C* subsumes *E, noted $C \prec_c E$, iff there exists a substitution θ such that the definite clause $tc(\vec{X}) \leftarrow L_h(\vec{X}, \vec{Y})$ substituted by θ, θ-subsumes E, and theta entails the constraint part $L_c(\vec{X}, \vec{Y})$: $C \prec_c E$ iff there exists θ s.t.*

$$tc(\vec{X})\theta = head(E) \text{ and } L_b(\vec{X}, \vec{Y})\theta \subseteq body(E) \text{ and } \mathcal{D} \models (\theta \rightarrow L_c(\vec{X}, \vec{Y}))$$

A constrained clause that subsumes the positive example Ex is termed complete.

We then define the consistency of a constrained clause with respect to a negative example:

Definition 3.2 *Let C be a constrained clause $C : tc(\vec{X}) \leftarrow L_h(\vec{X}, \vec{Y}) \wedge L_c(\vec{X}, \vec{Y})$ and let Ce be a clause. C is* inconsistent *with Ce iff there exists a pair of substitutions σ, ρ such that $L_h(\vec{X}, \vec{Y})$ substituted by σ, subsumes the body of Ce substituted by ρ, $L_c(\vec{X}, \vec{Y})$ is consistent with σ, and the head of C substituted by σ, is the negation of the* *head of Ce, substituted by ρ. Substitution σ is then termed* inconsistent wrt Ce, *noted $C\sigma \perp Ce$. $C\sigma \perp Ce$ iff there exists ρ s.t.*

$$L_h(\vec{X},\vec{Y})\sigma \subseteq body(Ce)\rho \text{ and } head(C)\sigma = \neg head(Ce)\rho \text{ and } \mathcal{D} \models (\exists)(L_c(\vec{X},\vec{Y}) \wedge \sigma)$$

C is consistent with, or discriminates, *Ce, if there exists no substitution σ inconsistent with Ce.*

Assumption : The positive example Ex is assumed consistent, according to the above definition, with any negative example Ce_i. It follows that the G set of solutions is not empty (since Ex itself, being both complete and consistent, is solution).

3.1.3 The search space

Let $\mathcal{C}$ denote, as in [30], the clause built from Ex by turning every occurrence of a term t_i in Ex into a distinct variable X_j; by construction, $Ex = \mathcal{C}\theta$, where θ denotes the substitution $\{X_j/t_i\}$:

$$\begin{array}{rl} \mathcal{C}: & fail(X,Y) \leftarrow uses(Z,T,U),\ speed(V,W),\ age(R,S) \\ Ex = \mathcal{C}\theta, & \text{with } \theta = \{X/a, Y/tom, Z/a, T/tom, U/5, V/a, W/120, R/a, S/20\} \end{array}$$

We restrict ourselves to exploring constrained clauses $tc(\vec{X}) \leftarrow L_h(\vec{X},\vec{Y}), L_c(\vec{X},\vec{Y})$, where L_h is a subset of the body of $\mathcal{C}$, and L_c only involves constraints of the types given in 2.4, e.g. domain constraints such as $(Y = tom)$, $(Y \neq ann)$ or $(S \in [20,50])$, and binary constraints such as $(X = Z)$, $(Y \neq T)$ or $(S - U \in [0,10])$.

3.1.4 Positive and Negative Substitutions

Let $gen(\mathcal{C})$ denote a generalization of $\mathcal{C}$ obtained by dropping literals in the body of $\mathcal{C}$. Substitutions on $gen(\mathcal{C})$ can be trivially embedded into substitutions of $\mathcal{C}$, by substituting variables other than that of $gen(\mathcal{C})$, by themselves. ¿From now on, no distinction is made between substitutions on a generalization of $\mathcal{C}$ and that on $\mathcal{C}$.
This enables to express consistency and completeness requirements with respect to particular substitutions of $\mathcal{C}$: substitution θ with $Ex = \mathcal{C}\theta$, termed *positive substitution*, and substitutions on some generalization of $\mathcal{C}$ inconsistent with some Ce_i, termed *negative substitutions* derived from Ce_i.

Table 1 shows the set of positive and negative substitutions corresponding to the previous examples. Clause $fail(X,Y) \leftarrow uses(Z,T,U), age(R,S)$ is inconsistent with Ce_1 according to negative substitutions σ_1 and σ_2 (respectively matching literal $uses(Z,T,U)$ with literals $uses(X,Y,10)$ and $uses(X,Z,0)$ in Ce_1).
Similarly, clause $fail(X,Y) \leftarrow uses(Z,T,U), speed(V,W)$ is inconsistent with Ce_2 according to negative substitution σ_3.

	X	Y	Z	T	U	V	W	R	S
θ	a	tom	a	tom	5	a	120	a	20
σ_1	$sk1$	$sk2$	$sk1$	$sk2$	10	–	–	$sk1$	10
σ_2	$sk3$	$sk4$	$sk3$	$sk5$	0	–	–	$sk3$	10
σ_3	b	ann	b	ann	$sk6$	b	0	–	–

Table 1: Positive and Negative substitutions on $\mathcal{C}$

Variables non involved in $\sigma_1, \sigma_2, \sigma_3$ are indicated by $'-'$ in Table 1. Variables in $\sigma_1, \sigma_2, \sigma_3$ are skolemized in order to keep track of the links between variables; these will enable the building of binary constraints (section 3.4).

Induction starts from a clause with an empty body ($fail(X, Y) \leftarrow$.) and specializes the body of this clause in order to find all constrained clauses complete and consistent. We first consider the specialization based on literals, then on domain constraints (section 3.3) and binary constraints (section 3.4).

3.2 Literal-based discrimination

Let g denote a generalization of $\mathcal{C}$ obtained by dropping literals in the body of $\mathcal{C}$. Then, g is complete since $\mathcal{C}$ subsumes Ex by construction.

The consistency of such g with respect to negative examples can be expressed as a boolean discrimination problem: g discriminates Ce iff the body of g involves a literal built on a predicate symbol that does not occur in Ce.

More formally, let NoP be the number of predicates in $\mathcal{C}$ and let the set of predicates in g be described by the boolean vector $(P_1, \ldots, P_{NoP})$, where P_i is true iff predicate p_i occurs in g. Let A_j denote the indices of predicates *absent* in negative example Ce_j. Then,

$$(g \text{ discriminates } Ce_j) \qquad \text{iff} \qquad (\bigvee_{i \,\in\, A_j} P_i)$$

In our example, a generalization of $\mathcal{C}$ discriminates Ce_1 iff it includes predicate *speed*, for it is the only predicate in $\mathcal{C}$ that does not occur in Ce_1.

Consider now the case of a negative example Ce not discriminated by g through literals; then from definition 3.2, there exists at least one substitution σ on g, that is inconsistent with Ce. We address the discrimination of a negative substitution by means of constraints.

3.3 Domain Constraint-based Discrimination

Let g be a constrained clause such that there exists a unique substitution σ on g inconsistent with negative example Ce. We now explore how to remove this inconsistency, by adding to the constraint part of g a domain constraint that makes negative substitution σ no longer satisfiable.

Note that such domain constraint, in order for g to be still complete, must be entailed by positive substitution θ.

Definition 3.3 *An* elementary discriminant constraint*, with respect to negative substitution σ and variable X, is a domain constraint set on X that is entailed by positive substitution θ and makes σ unsatisfiable, or* discriminates σ.
An elementary discriminant constraint wrt X and σ that is maximally general is termed maximally discriminant.

The building of elementary discriminant constraints can be brought back to an attribute-value discrimination problem:

Let $(X \in dom(X))$ be a domain constraint, where $dom(X)$ is strictly included in the initial domain of instantiation of X, noted Ω_X. Then, this constraint is entailed by θ iff $X.\theta$ is ground, and belongs to $dom(X)$. Conversely, this constraint discriminates σ iff $X.\sigma$ is ground and does not belong to

$dom(X)$.

We restrict ourselves to three kinds of variables, linear, tree-structured, and nominal variables. This restriction, which meets most real-world problems, together with our restricted CLP language (sec. 2.4), ensures that there exists at most one maximally discriminant domain constraint wrt a substitution σ and a variable X, noted $(X \in dom_\sigma(X))$:

- if X is linear (real or integer-valued), there exists a maximally discriminant constraint on X wrt σ iff $X.\theta$ and $X.\sigma$ are distinct constants, in which case $dom_\sigma(X)$ is the largest interval including $X.\theta$ and excluding $X.\sigma$.
  $S \in]10, \infty[$ is the maximally discriminant constraint on variable S that discriminates σ_1 (in Table 1).

- if X is tree-structured, there exists a maximally discriminant constraint on X wrt σ iff $X.\theta$ and $X.\sigma$ are distinct constants with $X.\theta \not\leq X.\sigma$, in which case $dom_\sigma(X)$ is the subtree the root of which is the most general value covering $X.\theta$ and not covering $X.\sigma$.

- if X is nominal, there exists a maximally discriminant constraint on X wrt σ iff $X.\theta$ and $X.\sigma$ are distinct constants, in which case, $dom_\sigma(X)$ is the complementary of $X.\sigma$ in Ω_X. The corresponding maximally discriminant constraint is then conveniently expressed as $(X \neq X.\sigma)$.
  $(Y \neq ann)$ is the maximally discriminant constraint on variable Y that discriminates σ_3 in Table 1.

If $dom_\sigma(X)$ exists, X is said *discriminant* with respect to σ; otherwise $dom_\sigma(X)$ is set to the empty set.

The building of maximally discriminant domain constraints is illustrated on our toy example. The initial domains of variables are given as:

$\Omega_X = \Omega_Z = \Omega_V = \Omega_R = \{a, b, c\}$	*objects that can fail*
$\Omega_Y = \Omega_T = \{tom, bill, ann\}$	*the users*
$\Omega_U = \Omega_S = [0, 30]$	*age and period of use of an object*
$\Omega_W = [0, 180]$	*speed of an object*

	X	Y	Z	T	U	V	W	R	S
σ_1	–	–	–	–	$U = [0, 10[$	–	–	–	$S =]10, 180]$
σ_2	–	–	–	–	$U =]0, 30]$	–	–	–	$S =]10, 180]$
σ_3	$X \neq b$	$Y \neq ann$	$Z \neq b$	$T \neq ann$	–	$V \neq b$	$W =]0, 180]$	–	–

Table 2: Maximally Discriminant Domain Constraints associated to Negative Substitutions

Symbol $'-'$ at row i and column j indicates that variable j is not discriminant wrt substitution σ_i. Table 2 can be read as: row j enumerates all ways of discriminating negative substitution σ_j via domain constraints; such discrimination of σ_j is thus given by the disjunction of these maximally discriminant domain constraints. More formally,

Proposition 1

Let g denote a constrained clause in our search space, where the constraint part is only composed of domain constraints $(X \in dom(X))$, and such that g subsumes Ex. Let σ be a negative substitution. Then

$$(g \textit{ discriminates } \sigma) \quad \textit{iff} \quad \Big(\bigvee_{X \textit{ discriminant wrt } \sigma} (dom(X) \subset dom_\sigma(X)) \Big)$$

Assuming that tree-structured and linear domains can be explored with a bounded cost, the complexity of discriminating a negative substitution via domain constraints is then in $\mathcal{O}(|var(\mathcal{C})|)$, where $|var(\mathcal{C})|$ denotes the number of variables in $\mathcal{C}$.

3.4 Discriminant binary constraints

The main motivation for binary constraints is to enable more compact generalizations, whenever discriminant features actually involve a correlation between variables.

The binary constraint based discrimination is brought back to domain constraint based discrimination, by considering auxiliary variables in the line of [13], termed *relational variables*. Relational variables encode the relationship between the instantiations of two variables; this enables to handle binary constraints through domain constraints defined on relational variables.

Let X and Y be two variables with same initial domain of instantiation (this information is to be supplied by the user). Let variable $X \star Y$ be valued in $\Omega_{X\star Y} = \mathbb{R}$ (respectively $\{true, false\}$) when X and Y are linear (resp. nominal or tree-structured) variables. For any substitution ρ on $\mathcal{C}$, let $(X \star Y).\rho$ be defined as follows:

- if X and Y are tree-structured or nominal variables, then $(X \star Y).\rho$ is $true$ iff X and Y are substituted by the same term through ρ, false iff $X.\rho$ and $Y.rho$ are distinct constants, and non-ground, otherwise.
- if X and Y are linear variables, then $(X \star Y).\rho$ is 0 iff X and Y are substituted by the same term through ρ, $(X \star Y).\rho$ is $X.\rho - Y.\rho$ iff $X.\rho$ and $Y.rho$ are distinct constants, and non-ground, otherwise.

By construction, any binary constraint on variables X and Y in the considered language (section 2.4) can be described as a domain constraint $(X \star Y \in dom(X \star Y))$.

Relational variables are represented via additional columns in the tabular representation of substitutions (Table 3); non grounded relational variables are indicated by $'-'$.

	$X\star Z$	$X\star V$	$X\star R$	$Z\star V$	$Z\star R$	$V\star R$	$Y\star T$	$U\star S$
θ	$true$	$true$	$true$	$true$	$true$	$true$	$true$	-15
σ_1	$true$	$-$	$true$	$-$	$true$	$-$	$true$	0
σ_2	$true$	$-$	$true$	$-$	$true$	$-$	$false$	-10
σ_3	$true$	$true$	$-$	$true$	$-$	$-$	$true$	$-$

Table 3: Relational variables of Positive and Negative substitutions

As in section 3.3, maximally discriminant binary constraints noted $dom_\sigma(X \star Y)$ can be associated to any negative substitution σ (Table 4).

	$X\star Z$	$X\star V$	$X\star R$	$Z\star V$	$Z\star R$	$V\star R$	$Y\star T$	$U\star S$
σ_1	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$U < S$
σ_2	$-$	$-$	$-$	$-$	$-$	$-$	$Y = T$	$U < S - 10$
σ_3	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$

Table 4: Corresponding Maximally Discriminant Binary Constraints

Symbol $'-'$ at row i and column j indicates that relational variable j is not discriminant wrt substitution σ_i. Each row in Table 4 describes all ways of discriminating the corresponding negative substitution via binary constraints; the discrimination of this substitution is given by the disjunction of these maximally discriminant binary constraints.

A straightforward extension of Proposition 1 is:

Proposition 2
Let g denote a constrained clause that g subsumes Ex ($g \prec_c Ex$). Let σ be a negative substitution. Let var_σ denote the set of initial and relational variables discriminant with respect to σ. Then

$$(g \textit{ discriminates } \sigma) \qquad \textit{iff} \qquad \Big(\bigvee_{X \,\in\, var_\sigma} (dom(X) \subset dom_\sigma(X)) \Big)$$

The complexity of discriminating a negative substitution via domain and binary constraints is then in $\mathcal{O}(|var(\mathcal{C})|^2)$.

Note that only initial and relational variables that are grounded by θ can be discriminant and need therefore be explored. Let NoX be the number of such variables, and let $\{X_1, ..X_{NoX}\}$ denote the set of such variables.

3.5 Discriminant Induction and Constraint Satisfaction

Let us consider a constrained clause g in the search space. ¿From what precedes, g is described by the set of predicates occurring in g (noted as $(P_1, ..P_{NoP}) \in \{0,1\}^{NoP}$, where NoP is the number of predicates in $\mathcal{C}$) and the domain constraints $dom(X_i)$ associated to variable $X_i, i = 1..NoX$; the default value for $dom(X_i)$ is the initial domain of X_i, noted Ω_{X_i}.
More formally, let Ω denote the cross product of $\{0, 1, \{0, 1\}\}^{NoP}$ and the power sets of Ω_{X_i}, for $i = 1..NoX$:

$$\Omega = \{0, 1, \{0, 1\}\}^{NoP} \times \mathcal{P}(\Omega_{X_1}) \times \ldots \times \mathcal{P}(\Omega_{X_{NoX}})$$

Constrained clause g is mapped onto the element of Ω noted ω_g, with $\omega_g = (P_1, \ldots, P_{NoP}, dom(X_1), \ldots, dom(X_{NoX})$. The inverse mapping is defined from Ω onto the induction search space, as follows:

$$\begin{array}{lcl} \Omega & \rightarrow & \mathcal{L}_g \\ \omega = (P_1, \ldots, P_{NoP}, V_1, \ldots, V_{NoX}) & \rightarrow & g_\omega \end{array}$$

where the body of g_ω is built of (a) constraints $(X_i \in V_i)$, for every variable X_i such that $V_i \neq \Omega_{X_i}$; (b) literals in $\mathcal{C}$ involving the variables in the above constraints; and (c) one literal built on any predicate p_i such that P_i is 1 and which is not yet in the body of g_ω.

What has been done in the previous steps allows to express the completeness and consistency of a constrained clause g via new constraints, termed *cp-constraints*, set on ω_g. These cp-constraints formalize the induction goal in terms of constraint programming: in this sense, they are at some meta-level with respect to the constraints involved in the solutions of induction.

By recollecting results of sections 3.3 and 3.4, the completeness of g is equivalent to cp-constraint c_0, with:

$$c_0 \;=\; \bigwedge_{i=1}^{NoX} X_i.\theta \in dom(X_i) \tag{1}$$

The consistency requirement is decomposed as follows. The consistency of g with respect to negative example Ce_j is ensured by means of literal based-discrimination (section 3.2), otherwise all negative substitutions derived from Ce_j must be discriminated. The discrimination of negative substitution σ_j by a constraint set on variable X_i, denoted c_{ij}, is given by:

$$c_{ij} \;=\begin{cases} dom(X_i) \subset dom_{\sigma_j}(X_i) & \text{if } X_i \text{ is discriminant wrt } \sigma_j \\ false & \text{Otherwise} \end{cases}$$

Then the discrimination of σ_j by g, denoted $c(\sigma_j)$, is the disjunction of c_{ij} over X_i in $X_1, ..X_{NoX}$:

$$c(\sigma_j) = \qquad \bigvee_{i=1}^{NoX} c_{ij}$$

Last, let Σ_j be the set of negative substitutions derived from Ce_j. The consistency of g with respect to Ce_j is then equivalent to:

$$(\bigvee_{i\; \in\; A_j} P_i) \bigvee (\bigwedge_{\sigma_k\; \in\; \Sigma_j} c(\sigma_k))$$

Finally, let $\Sigma = \{\sigma_1, \ldots, \sigma_{NoL}\}$ be the set of all negative substitutions; any σ_j in Σ is derived from a given negative example Ce_k ; let Ab_j denote the indices of predicates not occurring in Ce_k. Then, the consistency of g is equivalent to the conjunction, over all negative substitutions σ_j, of cp-constraints c_j defined as:

$$c_j = \qquad (\bigvee_{i \in Ab_j} P_i) \bigvee (\bigvee_{i=1}^{NoX} c_{ij}) \tag{2}$$

Let $\mathcal{CS}$ denote the set (conjunction) of cp-constraints $\{c_0,\ c_1, \ldots, c_{NoL}\}$. By construction,

Proposition 3
Constrained clause g is complete and consistent iff ω_g satisfies the set of cp-constraints $\mathcal{CS}$.

$$(g \textit{ is complete and consistent}) \qquad \textit{iff} \qquad (\omega_g \textit{ satisfies } \mathcal{CS})$$

In this sense, the constraint satisfaction problem $\mathcal{CS}$ is *equivalent* to the discriminant induction problem in the considered CLP language.
Note that $\mathcal{CS}$ is *satisfiable*, since it admits at least the solution associated to $g = Ex$.

3.6 How Constraint Solving can help Discriminant Induction

Based on the above reformulation of the discriminant induction problem, we present an algorithm named *ICP* (for *Inductive Constrained Programming*), that builds the set of all maximally discriminant constrained clauses.

The set of solutions G is initialized to the empty set; the set of active constraints, noted $\mathcal{CS}_{act}$, is initialized to $\mathcal{CS}$.

Init

$G = \phi$.
$\mathcal{CS}_{act} = \mathcal{CS} = \{c_1, c_2, \ldots c_{NoL}\}$
* Prune $\mathcal{CS}_{act}$.
$\omega = (P_1, \ldots P_{NoP}, dom(X_1), \ldots dom(X_{NoX})) = (\{0,1\}, \ldots \{0,1\}, \Omega_{X_1}, \ldots \Omega_{X_{NoX}})$.

Build G

While $\mathcal{CS}_{act}$ is not empty,

Select a constraint c_k in $\mathcal{CS}_{act}$.

If possible, * Specialize ω to satisfy c_k

Otherwise, Backtrack on the specialization choices

Let g_ω be the constrained clause associated to ω.

* If g_ω is maximally discriminant, $G = G \bigcup g_\omega$.

Backtrack on the specialization choices.

Specialize ω **to satisfy** c_k (non deterministic)

Select P_i such that $(P_i = 1)$ entails c_k *Literal-based specialization*

Do $P_i = 1$.

Remove from $\mathcal{CS}_{act}$ all c_m entailed by $(P_i = 1)$.

Constraint-based specialization

Or select a variable X_j the specialization of which is licit [1],

and such that $dom_{\sigma_k}(X_i) \neq \phi$

Do $dom(X_j) = dom(X_j) \bigcap dom_{\sigma_k}(X_i)$

Remove from $\mathcal{CS}_{act}$ all c_m entailed by the conjunction of $(X_i \in dom(X_i))$.

In this scheme, constraint solving is employed to several tasks:

It can first be used to set the constraint satisfaction problem $\mathcal{CS}$, from the positive and negative substitutions (Tables 1 and 3): maximally discriminant constraint $dom_\sigma(X)$ (in Tables 2 and 4) are built as maximally general solutions (in the considered language) to cp-constraint $(X.\theta \subset V) \wedge (X.\sigma \bigcap V = \phi) \wedge (V \subset \Omega_X)$.

It allows to prune the set of constraints: a partial order on the negative substitutions (respectively examples) with respect to the positive substitution (resp. example) can be defined [30]. Minimal substitutions and examples with respect to this partial order, can be viewed as "nearest-misses": all substitutions and examples but the minimal ones, can be soundly pruned. This pruning was explicitly dealt with in previous works [27, 28, 30]; it turns out to be a special case of constraint entailment. Checking whether a cp-constraint c_i entails a cp-constraint c_j (and removing c_j if it is the case) can therefore be delegated to constraint propagation.

It allows of course, to solve $\mathcal{CS}$ (routine *Specialize*), through selecting specialization choices, propagating these choices in order to update $\mathcal{CS}_{act}$, and backtracking.

Last, it allows for testing whether g_ω is maximally general: the fact that ω is a maximally general solution of $\mathcal{CS}$ can be expressed via additional cp-constraints, which cannot be detailed here because of space limitations.

The CSP corresponding to our toy example is represented in Table 5 (not all variables are represented given the allowed width). Columns 2-4 are respectively associated to predicates *uses, speed* and *age*, noted P_u, P_s and P_a.

---

[1]Setting a constraint on an initial variable X_i requires the atom involving this variable in $\mathcal{C}$, to occur in the body of the solution. The predicate this literal is built on, termed *parent predicate* of X_i, therefore occurs in the solution too. One can similarly define the parent predicate(s) of a relational variable. Setting a constraint on a variable is said to be *licit* whenever its parent predicates have been selected during the literal-based specialization.

	P_u	P_s	P_a	W	$Y \star T$	$U \star S$
c_1	$-$	1	$-$	$-$	$-$	$U < S$
c_2	$-$	1	$-$	$-$	$Y = T$	$U < S - 10$
c_3	$-$	$-$	1	$W =]0, 180[$	$-$	$-$
ω	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$	$dom(W) =]0, 180[$	$dom(Y \star T) = true$	$dom(U \star S) = [-30, 0[$

Table 5: System $\mathcal{CS}$ and a Solution

The corresponding constrained clause g_ω is then :

$$fail(X,Y) \leftarrow uses(Z,T,U), speed(V,W), age(R,S), (Y = T), (W > 0), (U < S)$$

If the constraints $(X = Z), (X = V), (X = R)$ were added, this clause would be interpreted that any X fails to be driven by Y, if X is a moving stuff $(speed(X,W), (W > 0))$ and Y is a second hand user of X $(uses(X,Y,U), age(X,S), (U < S))$.

3.7 Small induction and lazy learning

This approach leads to a 2-step scheme of induction:
During the first step, termed *small induction*, the set of constraints $\mathcal{CS}$ is built from the initial learning problem (3.5). During the second step, CSP $\mathcal{CS}$ is solved and maximally discriminant clauses are derived from the solutions of $\mathcal{CS}$ (algorithm *ICP*).

The advantage of this scheme is twofold. First, induction may have efficient external tools in charge of the burden of search. The only impact of this, w.r.t. induction (besides the expected computational speed-up) is to free the user from supplying ad hoc additional knowledge.

Second, some lazy learning can easily take place in this scheme, as follows. It is straightforward to show that:

Proposition 4
Let E be a conjunction of atoms. Then E implies the target predicate tc according to G, iff E can be put as $\mathcal{C}.\rho$ and ρ entails $\mathcal{CS}$.

$$(G, E \models TC) \qquad \textit{iff} \qquad \textit{there exists } \rho \ / \ (E = \mathcal{C}.\rho) \textit{ and } (\rho \textit{ entails } \mathcal{CS})$$

where $TC = tc(\vec{X})\rho$.
Small learning can then be viewed as a lazy, on-fly learning: it produces $\mathcal{CS}$, which can be viewed as a computational characterization of G and does not need to be solved in order to classify unseen example. Furthermore this classification can be justified [29].

3.8 Complexity

The number of constrained variables in $\mathcal{CS}$ is upper bounded by the number NoP of predicates, plus the number NoX of initial and relational variables grounded by θ. The number of constraints in $\mathcal{CS}$ is upper bounded by the number of negative substitutions NoL. If J denotes an upper bound on the number of literals built on a given predicate in the examples, then NoL is in $J^{NoP \times J}$ ([30]).

The number of possible solutions, i.e. the size of G, is then in $\mathcal{O}((NoP + NoX)^{NoL})$. No wonder: the size of G is known to be exponential in the number of literals even in the propositional frame [8].

Now consider the complexity of checking whether a given example E satisfies G: from Proposition 4, it is proportional to the number of substitutions ρ such that $E = \mathcal{C}\rho$, and to the product of the number of constrained variables $NoP+NoX$ and the number of constraints $1+NoL$ in $\mathcal{CS}$. The classification complexity is then in $\mathcal{O}(NoL^2 \times (NoP+NoX))$. As expected from the propositional case [9], the computational characterization of G given by $\mathcal{CS}$ is significantly more tractable than the explicit building of G.

4 Discussion and Perspectives

This section first discusses the choice of maximally discriminant induction, made in this work. Then, it situates the proposed cooperation of ILP and CLP, with respect to some previous works devoted to generalization of constraints [21, 16] or reformulation of ILP problems [13, 7]. The limitation to a unique positive example, together with the problem of noisy data, is then discussed.

4.1 Redundant Learning

The presented work differs from all ILP approaches we are aware of, in that it aims at characterizing *all* maximally discriminant clauses covering a given example. In contrast, many learners are concerned with finding specific generalizations [18, 24, 20], or *one* discriminant generalization of an example [22, 3, 19], which can be transformed into an element of G via a post-pruning phase.

The issue of redundant learning was motivated in [29]: redundant learning ensures robust learning when dealing with noisy data, or data that imperfectly encode the underlying problem. The point of noise can be addressed via MDL-based heuristics, as in FOIL [23]; but these heuristics apply for ground examples only. The latter point makes any selection of the candidate solutions be potentially harmful. This is why we investigate the building of the whole G set of solutions, in the line of Version Spaces, rather than seeking some 'best' solution according to some criterion.

4.2 Generalization from constraints

As far as we know, the generalization from constraints has only been addressed so far by Page and Frisch [21] and Mizoguchi and Ohwada [16].

In [21], the goal is to generalize constrained atoms. Constrained atoms are handled as definite clauses whose antecedents express the constraints. Constrained generalizations of two atoms are built from the sorted generalizations defined on their arguments. In both [21] and our approach, generalization ultimately proceeds by building constraints. But different issues are addressed. In [21], the main difficulty arises from the possibly multiple generalizations of two terms, which does not occur in our restricted language (section 3.3). In opposition, the main difficulty here comes from the multiple structural matchings among examples (section 3.8) while such matching uniquely follows from the unique atom considered in [21].

Another approach of the generalization of constrained clauses is presented by Mizoguchi and Ohwada [16]. This work is nicely motivated by geometrical applications (avoiding the collision between objects and obstacles). The region of safe moves of an object can be 'naturally' described through a set of linear constraints; the goal consists in automatically acquiring such constraints from examples.

[16] first extend the definition of some typical induction operators (minimal generalization, absorption, lgg) to constrained clauses. Then, an *ad hoc* domain theory being given, examples are described by constrained atoms which are generalized through absorption and lgg, in the line of [18].

In what regards the roles respectively devoted to ILP and CLP, the essential differences can be summarized as follows: the induction of constrained clauses is done (a) by incorporating the structure of constraints into ILP, in [21]; (b) by extending the inverse resolution approach to CLP in [16]; and by interleaving ILP and CLP in our approach.

4.3 Reformulation

A strong motivation for reformulating ILP problems into simpler problems, e.g. in propositional form, is that propositional learners are good at dealing with numbers [13, 6, 7]. LINUS [13] achieves such transformation under several assumptions, which altogether ensure that one first-order example is transformed into *one* attribute-value example; this transformation thereby does not address the case of multiple structural matchings among examples. LINUS nicely uses the theory of the domain in order to introduce new variables and enrich the attribute-value representation of the examples.

Another reformulation approach, based on *morions*, is presented by Zucker and Ganascia [7]. The goal consists in focusing generalization by specifying the wanted level of description of the examples. Morions express the entities of the problem domain. A morion of a given level can be decomposed into one or several morions of a lower level (e.g. the description of the morion *car* involves the description of the four *tire* morions). A moriological reformulation factually corresponds to a selection of the predicates *and the matchings among literals built on these predicates* to take into account during generalization.

Note that [13] and [7] both map an induction problem into another induction problem. In opposition, the mapping presented here enables a shift of paradigm: an induction problem is transformed into a constraint satisfaction problem, which can in turn be solved by an external tool.

4.4 Applicability

Only one positive example has been considered so far. The general case is addressed by successively generalizing one current example, against other examples [29]: this amounts to embedding the Version Space approach into a Star-like approach [15].

For any example seed Ex, let the star of Ex be the G set of maximally discriminant constrained clauses generalizing the seed against all examples Ce_i not belonging to the same target concept, termed *G-star* and noted $G(Ex)$.

If n denotes the number of examples, the complexity of generalization then amounts to n times the complexity of building a G-star (3.8).

The complexity of classification similarly increases by a factor n: In case of discrimination among several classes (or antagonistic target predicates), the diagnosis associated from a new incoming E is determined by a majority vote among the training examples Ex_i, such that E belongs to the G-star $G(Ex_i)$.

Furthermore, this classification process permits to deal with noisy examples via simple counting-based heuristics, same as for propositional learning: only significant G-stars, that cover more than a given number of examples, are considered; furthermore, we stipulate an example to belong to a G-star, if it satisfies a given percentage of the constraints (rather than all constraints). Preliminary results on propositional problems are quite good [29].

4.5 Conclusion and Perspectives

It is emphasized that the presented approach *does not* allow to learn clauses that were beyond reach for state-of-the-art learners [23, 19]: as quoted before [26], *all concepts and operations of interest in [the] underlying domain of computation* [can] *be explicitly encoded in the form of Herbrand terms and operations on them.* Rather, it allows to learn numerical, unary or binary, constraints *without requiring any additional background theory* to be supplied.

This work extends the *tractable, computable* characterization of the set of solutions formulated by Hirsh in the propositional case [9], to constrained clauses, and includes some ways to handle noisy data. It opens several perspectives of research:

New variables (as in [13]) and new types of constraints could be considered. Ideally, language bias would be expressed via additional constraints (for instance, requiring the solution clauses to be connected could be expressed via additional constraints).

In what regards the resolution of $\mathcal{CS}$, first experimentations were done with Eclipse, based on Prolog. More powerful constraint solvers must be experimented in order to check the size limitations of the proposed learner.

Last, the user could supply some optimality function in order to guide the selection of the admissible solutions. Selective discriminant induction could then be reformulated as a constrained optimization problem (finding the optimum of the objective function still satisfying the constraints).

Acknowledgments

Thanks to J.F. Puget, who originated the idea of integrating CLP within ILP and who provided us with the relevant pointers in the CLP literature. The work of the authors has been partially supported by the ESPRIT BRA 6020 Inductive Logic Programming.

References

[1] The clp(r) language and system. *ACM Transactions on Programming Languages*, 14:339–395, 1992.

[2] G. Bisson. Kbg: A knowledge base generalizer. In B. Porter and R. Mooney, editors, *Proceedings of the seventh International Conference on Machine Learning*, pages 9–15. Morgan Kaufmann, 1990.

[3] M. Botta and A. Giordana. Smart+: A multi-strategy learning tool. In *Proceedings of IJCAI-93*, pages 937–943. Morgan Kaufmann, 1993.

[4] A. Colmerauer. An introduction to prolog iii. *CACM*, 33:69–90, 1990.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, and A. Aggoun. The constraint logic programming language chip. pages 249–265, 1988.

[6] S. Dzeroski, L. Todorovski, and T. Urbancic. Handling real numbers in ilp: a step towards better behavioral clones. In N. Lavrac and S. Wrobel, editors, *Proceedings of ECML-95, European Conference on Machine Learning.* Springer Verlag, 1995.

[7] J.-D. Zucker et J.-G. Ganascia. Selective reformulation of examples in concept learning. In W. Cohen and H. Hirsh, editors, *Proc. of ICML-94*, pages 352–360. Morgan Kaufmann, 1994.

[8] D. Haussler. Quantifying inductive bias: Ai learning algorithms and valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[9] H. Hirsh. Polynomial-time learning with version spaces. In *Proceedings of National Conference on Artificial Intelligence*, 1992.

[10] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proc. of the fourteenth ACM Symposium on the Principles of Programming Languages*, pages 111–119, 1987.

[11] J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, pages 503–581, 1994.

[12] J.-L. Lassez, M.J. Maher, and K. Mariot. Unification revisited. In H. Gallaire and J. Minker, editors, *Foundations of Deductive Databases*, pages 293–322. Morgan Kaufmann, 1987.

[13] N. Lavrac, S. Dzeroski, and M. Grobelnick. Learning non recursive definitions of relations with linus. In *Proceedings of EWSL'91*, 1991.

[14] J.W. Lloyd. *Foundations of Logic Programming, second extended edition.* 1987.

[15] R.S. Michalski. A theory and methodology of inductive learning. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 1. Morgan Kaufmann, 1983.

[16] F. Mizoguchi and H. Ohwada. Constraint-directed generalizations for learning spatial relations. In *Proceedings of ILP-91, International Workshop on Inductive Logic Programming*, 1991.

[17] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19:629–679, 1994.

[18] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory.* Ohmsha, Tokyo, Japan, 1990.

[19] S. Muggleton and A. Srinivasan. Progolxx. In S. Wrobel, editor, *Proceedings of ILP-94, International Workshop on Inductive Logic Programming*, 1994.

[20] C. Nedellec. How to specialize by theory refinement. In B. Neumann, editor, *Proceedings of ECAI*, pages 474–478. Wiley, 1992.

[21] C.D. Page and A.M. Frisch. Generalization and learnability: A study of constrained atoms. In S. Muggleton, editor, *Proceedings of the first International Workshop on Inductive Logic Programming*, pages 29–61, 1991.

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

[23] J.R. Quinlan. In Hirsh H. Cohen W., editor, *Proceedings of ICML-94, International Conference on Machine Learning.* Morgan Kaufmann, 1994.

[24] L. De Raedt. *Interactive Theory Revision: an inductive logic programming approach.* Academic Press, 1992.

[25] G. Sablon and L. De Raedt. Forgetting and compacting data in concept learning. In *Proceedings of IJCAI-95.* Morgan Kaufmann, 1995.

[26] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[27] M. Sebag. Using constraints to building version spaces. In L. De Raedt and F. Bergadano, editors, *Proceedings of ECML-94.* Springer Verlag, April 1994.

[28] M. Sebag. A Constraint Based Induction Algorithm in FOL. In W. Cohen and H. Hirsh, editors, *Proc. of ICML-94*, pages 275–283. Morgan Kaufmann, 1994.

[29] M. Sebag. 2^{nd} order understandability of disjunctive version spaces. In *Ws on ML and Comprehensibility, IJCAI-95.* Report LRI, Université Paris-Sud, 1995.

[30] M. Sebag and C. Rouveirol. Induction of maximally general clauses compatible with integrity constraints. In S. Wrobel, editor, *Proceedings of ILP-94, International Workshop on Inductive Logic Programming*, 1994.

[31] P. Van Hentenryck and Deville Y. Constraint logic programming. In *Proceedings of POPL'97*, 1987.

[32] P. Van Hentenryck and Deville Y. Operational semantics of constraint logic programming over finite domains. In *Proceedings of PLILP'91*, 1991.