Towards Inductive Constraint Solving

Slim Abdennadher[1] and Christophe Rigotti[2*]

[1]Computer Science Department, University of Munich
Oettingenstr. 67, 80538 München, Germany
Slim.Abdennadher@informatik.uni-muenchen.de

[2]Laboratoire d'Ingénierie des Systèmes d'Information
Bâtiment 501, INSA Lyon, 69621 Villeurbanne Cedex, France
Christophe.Rigotti@insa-lyon.fr

Abstract. A difficulty that arises frequently when writing a constraint solver is to determine the constraint propagation and simplification algorithm. In previous work, different methods for automatic generation of propagation rules [5, 17, 3] and simplification rules [4] for constraints defined over finite domains have been proposed. In this paper, we present a method for generating rule-based solvers for constraint predicates defined by means of a constraint logic program, even when the constraint domain is infinite. This approach can be seen as a concrete step towards *Inductive Constraint Solving.*

1 Introduction

Inductive Logic Programming (ILP) is a machine learning technique that has emerged in the beginning of the 90's [12]. ILP has been defined as the intersection of inductive learning and logic programming. It aims at inducing hypotheses from examples, where the hypothesis language is the first order logic restricted to Horn clauses. To handle numerical knowledge, an inductive framework, called *Inductive Constraint Logic Programming* (ICLP), similar to that of ILP but based on constraint logic programming schemes have been proposed [13]. ICLP extends ideas and results from ILP to the learning of constraint logic programs. In this paper, we propose a method to *learn* rule-based constraint solvers from the definitions of the constraint predicates. We call this approach *Inductive Constraint Solving* (ICS). It extends previous works [5, 17, 3] where different methods for automatic generation of propagation rules for constraints defined over finite domains have been proposed.

In rule-based constraint programming, the solving process of constraints consists of a repeated application of rules. In general, we distinguish two kinds of rules: simplification and propagation rules. Simplification rules rewrite constraints to simpler constraints while preserving logical equivalence, e.g. $X{\leq}Y \wedge Y{\leq}X \Leftrightarrow X{=}Y$. Propagation rules add new constraints which are logically redundant but may cause further simplification, e.g. $X{\leq}Y \wedge Y{\leq}Z \Rightarrow X{\leq}Z$.

* The research reported in this paper has been supported by the Bavarian-French Hochschulzentrum.

In this paper, we present an algorithm, called PropMiner, that can be used to generate propagation rules for constraint predicates defined by means of a constraint logic program, even when the constraint domain is infinite. The PropMiner algorithm can be completed with the algorithm presented in [4] to transform some propagation rules into simplification rules improving both the time and space behavior of constraint solving.

The combination of these techniques can be seen as a true ICS tool. Using this tool, the user only has to determine the semantics of the constraints of interest by means of their intentional definitions (a constraint logic program), and to specify the admissible syntactic form of the rules he wants to obtain.

Example 1. Consider the following constraint logic program, where $min(A, B, C)$ means that C is the minimum of A and B:

$$\begin{aligned} min(A, B, C) &\leftarrow A{\leq}B \ \wedge \ C{=}A. \\ min(A, B, C) &\leftarrow B{\leq}A \ \wedge \ C{=}B. \end{aligned}$$

For the predicate min, our algorithm PropMiner described in Section 2 generates the following propagation rules if the user specifies that the left hand side of the rules may consist of min constraints and equality constraints:

$$\begin{aligned} min(A, B, C) &\Rightarrow C{\leq}A \ \wedge \ C{\leq}B. \\ min(A, B, C) \ \wedge \ A{=}B &\Rightarrow A{=}C. \end{aligned}$$

For example, the second rule means that the constraint $min(A, B, C)$ when it is known that the input arguments A and B are equal can propagate the constraint that the output C must be equal to the input arguments.

If the user additionally allows disequality and less-or-equal constraints on the left hand side of the rules, the algorithm generates the following rules:

$$\begin{aligned} min(A, B, C) \ \wedge \ C{\neq}B &\Rightarrow C{=}A. \\ min(A, B, C) \ \wedge \ C{\neq}A &\Rightarrow C{=}B. \\ min(A, B, C) \ \wedge \ B{\leq}A &\Rightarrow C{=}B. \\ min(A, B, C) \ \wedge \ A{\leq}B &\Rightarrow C{=}A. \end{aligned}$$

Using the algorithm presented in [4] some propagation rules can be transformed into simplification rules and we obtain the following rule-based constraint solver for min:

$$\begin{aligned} min(A, B, C) &\Rightarrow C{\leq}A \ \wedge \ C{\leq}B. \\ min(A, A, C) &\Leftrightarrow A{=}C. \\ min(A, B, C) \ \wedge \ C{\neq}B &\Rightarrow C{=}A. \\ min(A, B, C) \ \wedge \ C{\neq}A &\Rightarrow C{=}B. \\ min(A, B, C) \ \wedge \ B{\leq}A &\Leftrightarrow C{=}B \ \wedge \ B{\leq}A. \\ min(A, B, C) \ \wedge \ A{\leq}B &\Leftrightarrow C{=}A \ \wedge \ A{\leq}B. \end{aligned}$$

For example, the goal $min(A, B, B)$ will be transformed into $B{\leq}A$ using the first propagation rule and then the second last simplification rule. □

The generated rules can be directly encoded in a rule-based programming language, e.g. Constraint Handling Rules (CHR) [6] to provide a running constraint solver. The Inductive Constraint Solving tool presented in this paper can also be simply used as a software engineering tool to help solver developers to find out propagation and simplification rules.
The paper is organized as follows. In Section 2, we present an algorithm to generate propagation rules for constraint predicates defined by a constraint logic program. In Section 3, we give more examples for the use of our algorithm. We discuss in Section 4 how recursive programs can be handled. Finally, we conclude with a summary and compare the proposed approach with related work.

2 Generation of Propagation Rules

In this section, we present an algorithm, called PropMiner, to generate propagation rules for constraints using the intensional definitions of the constraint predicates. These definitions are given by means of a program in a constraint logic programming (CLP) language. We assume some familiarity with constraint logic programming as defined by Jaffar and Maher in [9] and follow their definitions and terminology when applicable.

The CLP programs are parameterized by a constraint system defined by a 4-tuple $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$ and a signature Π determining the predicate symbols defined by a program. Σ is a signature determining the predefined predicate and function symbols, $\mathcal{D}$ is a Σ-structure (the domain of computation), $\mathcal{L}$ is a class of Σ-formulas closed by conjunction and called constraints, and $\mathcal{T}$ is a first-order Σ-theory that is an axiomatization of the properties of $\mathcal{D}$.
We require that $\mathcal{D}$ is a model of $\mathcal{T}$ and that $\mathcal{T}$ is satisfaction complete with respect to $\mathcal{L}$, that is, for every constraint $c \in \mathcal{L}$ either $\mathcal{T} \models \tilde{\exists} c$ or $\mathcal{T} \models \neg\tilde{\exists} c$, where $\tilde{\exists}(\phi)$ denotes the existential closure of ϕ. Note that these requirements are fulfilled by most commonly used CLP languages.
In the rest of this paper, we use the following terminology.

Definition 1. A *constrained clause* is a rule of the form

$$H \leftarrow B_1 \wedge \ldots \wedge B_n \wedge C_1 \wedge \ldots \wedge C_m$$

where $H, B_1, \ldots, B_n$ are atoms over Π and $C_1, \ldots, C_m$ are constraints. A *goal* is a set of atoms over Π and constraints, interpreted as their conjunction. An *answer* is a set of constraints also interpreted as their conjunction. A *CLP program* is a finite set of constrained clauses. The logical semantics of a CLP program P is its Clark's completion and is denoted by P^*.

In programs, goals and answers, when clear from the context, we use upper case letters (resp. lower case and numbers) to denote variables (resp. constants).

2.1 Rules of Interest

A *propagation pattern* is a set of constraints and of atoms over Π, interpreted as their conjunction.
A *propagation rule* is a rule of the form $C_1 \Rightarrow C_2$ or of the form $C_1 \Rightarrow false$, where C_1 is a propagation pattern and C_2 is a set of constraints (also interpreted as their conjunction). C_1 is called the *left hand side* (lhs) and C_2 the *right hand side* (rhs) of the rule. A rule of the form $C_1 \Rightarrow false$ is called *failure rule*. To formulate the logical semantics of these rules, we use the following notation: let $\mathcal{V}$ be a set of variables then $\exists_{-\mathcal{V}}(\phi)$ denotes the existential closure of ϕ except for the variable in $\mathcal{V}$.

Definition 2. A propagation rule $\{c_1, \ldots, c_n\} \Rightarrow \{d_1, \ldots, d_m\}$ is *valid*[1] wrt. the constraint theory $\mathcal{T}$ and the CLP program P iff $P^*, \mathcal{T} \models \bigwedge_i c_i \rightarrow \exists_{-\mathcal{V}}(\bigwedge_j d_j)$, where $\mathcal{V}$ is the set of variables appearing in $\{c_1, \ldots, c_n\}$.
A failure rule $\{c_1, \ldots, c_n\} \Rightarrow false$ is *valid* wrt. $\mathcal{T}$ and P if and only if $P^*, \mathcal{T} \models \neg\tilde{\exists}(\bigwedge_i c_i)$.

To reduce the number of rules which are uninteresting to build a solver, we restrict with a syntactic bias the generation to a particular set of rules called *relevant propagation rules*. These rules must contain in their lhs atoms corresponding to the predicates on which we want to propagate information, and all elements in this lhs must be connected by common variables. This is defined more precisely by the notion of *interesting pattern*.

Definition 3. A propagation pattern $\mathcal{A}$ is an *interesting pattern* wrt. a propagation pattern $Base_{lhs}$ if and only if the following conditions are satisfied:

1. $Base_{lhs} \subseteq \mathcal{A}$.
2. the graph defined by the relation $join_{\mathcal{A}}$ is connected, where $join_{\mathcal{A}}$ is a binary relation that holds for pairs of elements in $\mathcal{A}$ that share at least one variable, i.e., $join_{\mathcal{A}} = \{\langle c_1, c_2 \rangle \mid c_1 \in \mathcal{A}, c_2 \in \mathcal{A}, Var(\{c_1\}) \cap Var(\{c_2\}) \neq \emptyset\}$, where $Var(\{c_1\})$ and $Var(\{c_2\})$ denote the variables appearing in c_1 and c_2, respectively.

A *relevant propagation rule* wrt. $Base_{lhs}$ is a propagation rule such that its lhs is an interesting pattern wrt. $Base_{lhs}$.

2.2 The PropMiner Algorithm

In this section, we describe the PropMiner algorithm to generate propagation rules from a program P expressed in a CLP language determined by $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$. The algorithm takes as input the program P, a propagation pattern $Base_{lhs}$ and a set of constraints $Cand_{lhs}$ (for which we already have a built-in solver). It generates propagation rules that are valid wrt. $\mathcal{T}$ and P, relevant wrt. $Base_{lhs}$ and such that their lhs are subsets of $Base_{lhs} \cup Cand_{lhs}$.

[1] The requirement made on CLP programs that $\mathcal{T}$ must be satisfaction complete is not sufficient to ensure the decidability of the propagation rule validity. However, it should be noticed that the soundness of the algorithm proposed in Section 2.2 is not based on such a decidability property.

Principle From an abstract point of view, the algorithm enumerates each possible lhs subset of $Base_{lhs} \cup Cand_{lhs}$ (denoted by C_{lhs}). For each C_{lhs} it computes a set of constraints noted C_{rhs} such that $C_{lhs} \Rightarrow C_{rhs}$ is valid wrt. $\mathcal{T}$ and P and relevant wrt. $Base_{lhs}$.

---

begin

Let $\mathcal{R}$ be an empty set of rules.
Let L be a list containing all non-empty subsets
of $Base_{lhs} \cup Cand_{lhs}$ in any order.

Remove from L any element C which is not an interesting pattern wrt. $Base_{lhs}$.
Order L with any total ordering compatible with the subset partial ordering
(i.e., for all C_1 in L if C_2 is after C_1 in L then $C_2 \not\subset C_1$).

```
while L is not empty do
    Let C_lhs be the first element of L and then remove C_lhs from L.
    Let A be the set of answers for the goal C_lhs wrt. the program P.
    if A is empty then
        add the failure rule (C_lhs => false) to R
        and remove from L each element C such that C_lhs ⊂ C.
    else
        if A is finite then
            compute the set of constraints C_rhs
            as the least general generalization (lgg) of A
            if C_rhs is not empty then
                add the rule (C_lhs => C_rhs) to R
            endif
        endif
    endif
endwhile

output R
```

end

Fig. 1. The PROPMINER Algorithm

For each C_{lhs}, the algorithm PROPMINER determines C_{rhs} by calling the CLP system to execute C_{lhs} as a goal and then

1. if C_{lhs} has no answer then it produces the failure rule $C_{lhs} \Rightarrow false$.

2. if C_{lhs} has a finite number of answers $\{Ans_1, \ldots, Ans_n\}$ then let C_{rhs} be the *least general generalization* (lgg) of $\{Ans_1, \ldots, Ans_n\}$ as defined by [15]. C_{rhs} is then in some sense the strongest constraint common to all answers as illustrated below (see Example 2). If C_{rhs} is not empty then the algorithm produces the rule $C_{lhs} \Rightarrow C_{rhs}$.

It is clear that these two criteria can be used only if all answers can be collected in finite time. The application of the algorithm to handle recursive programs leading to non-terminating executions is discussed in Section 4.
The algorithm is given in Figure 2.2. To simplify its presentation, we consider that all possible lhs are stored in a list. For efficiency reasons the concrete implementation is based on a tree and unnecessary candidates are not materialized. More details on the implementation are given in Section 2.4.
A particular ordering is used to enumerate the lhs candidates so that the more general lhs are tried before the more specific ones. Then, we use the following pruning criterion which improves greatly the efficiency of the algorithm: if a rule $C_{lhs} \Rightarrow false$ is generated then there is no need to consider any superset of C_{lhs} to form other rule lhs.
We now illustrate on the following example the basic behavior of the algorithm PropMiner. More uses of the algorithm are given in Section 3.

Example 2. Consider the following CLP program defining p and q:

$$\begin{aligned} p(X,Y,Z) &\leftarrow q(X,Y,Z). \\ p(X,Y,Z) &\leftarrow X{\leq}W \wedge Y{=}W \wedge X{>}Z. \\ q(X,Y,Z) &\leftarrow X{\leq}a \wedge Y{=}a \wedge Z{\neq}b. \end{aligned}$$

We use the algorithm to find rules to propagate constraints over propagation patterns involving p. Let $Base_{lhs} = \{p(X,Y,Z)\}$ and let for example $Cand_{lhs}$ be the set $\{X{\leq}Z,\ Y{=}a,\ Z{=}b\}$.
When the while loop is entered for the first time we have

$$\begin{aligned} L = \{\ & \{p(X,Y,Z)\},\ \{p(X,Y,Z),\ X{\leq}Z\},\ \{p(X,Y,Z),\ Y{=}a\}, \\ & \{p(X,Y,Z),\ Z{=}b\},\ \{p(X,Y,Z),\ X{\leq}Z,\ Y{=}a\},\ \{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\}, \\ & \{p(X,Y,Z),\ Y{=}a,\ Z{=}b\},\ \{p(X,Y,Z),\ X{\leq}Z,\ Y{=}a,\ Z{=}b\}\ \} \end{aligned}$$

Each element in L is executed in turn as a goal and the corresponding answers are collected and used to build a rule rhs. For example, $\{p(X,Y,Z),\ Z{=}b\}$ leads to a single answer $Ans_1 = \{X{\leq}W,\ Y{=}W,\ X{>}Z,\ Z{=}b\}$. The lgg is simply Ans_1 itself and we have the propagation rule $\{p(X,Y,Z),\ Z{=}b\} \Rightarrow \{X{\leq}W,\ Y{=}W,\ X{>}Z,\ Z{=}b\}$. For $\{p(X,Y,Z), X{\leq}Z\}$ we have again a single answer $\{X{\leq}a, Y{=}a, Z{\neq}b, X{\leq}Z\}$ and thus also a trivial lgg producing the rule $\{p(X,Y,Z), X{\leq}Z\} \Rightarrow X{\leq}a, Y{=}a, Z{\neq}b, X{\leq}Z\}$.
For the goal $\{p(X,Y,Z),\ Y{=}a\}$, the situation is different since we have the two following answers $Ans_1 = \{X{\leq}a, Y{=}a, Z{\neq}b\}$ and $Ans_2 = \{X{\leq}a, Y{=}a, X{>}Z\}$. The lgg which is based on a syntactical generalization is $\{X{\leq}a, Y{=}a\}$ and we have the rule $\{p(X,Y,Z), Y{=}a\} \Rightarrow \{X{\leq}a, Y{=}a\}$.

The situation may be more tricky. For example, the goal $\{p(X,Y,Z)\}$ have two answers $Ans_1 = \{X{\leq}a,\ Y{=}a,\ Z{\neq}b\}$ and $Ans_2 = \{X{\leq}W,\ Y{=}W,\ X{>}Z\}$ having no common element. Fortunately, the lgg corresponds in some sense to the least upper bound of $\{Ans_1, Ans_2\}$ wrt. the θ-subsumption ordering [15] (more precisely it represents the equivalence class of constraints that corresponds to this least upper bound). Thus, the lgg of $\{Ans_1, Ans_2\}$ is $\{X{\leq}E,\ Y{=}E\}$, where E is a new variable, and the algorithm produces the rule $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E, Y{=}E\}$. However, it should be noticed that the notion of lgg is not based on the semantics of the constraints in the set of answers. Thus, two sets of answers that are equivalent wrt. the constraint theory but not identical from a syntactic point of view will lead in general to different lgg's. As shown in sections 2.3 and 3, the user can partially overcome this difficulty by providing ad hoc propagation rules to take into account the constraint semantics.

The effect of the pruning criterion is straightforward. The goal $G = \{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\}$ has no answer and leads to the rule $\{p(X,Y,Z),\ X{\leq}Z,\ Z{=}b\} \Rightarrow false$. Then the element $\{p(X,Y,Z),\ X{\leq}Z,\ Y{=}a,\ Z{=}b\}$ that is a super set of G is simply removed from L and will not be considered to generate any rule.

Properties It is straightforward to see that the algorithm is complete in the sense that if $C_{lhs} \subseteq Base_{lhs} \cup Cand_{lhs}$ is an interesting pattern wrt. $Base_{lhs}$ and there is no $C \subset C_{lhs}$ such that $C \Rightarrow false$ is valid, then C_{lhs} is considered by the algorithm as a candidate to form the lhs of a rule.
To establish the soundness of the algorithm, we need the following results presented in [9].

Theorem 1. Let P be a program in the CLP language determined by $\langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$, where $\mathcal{D}$ is a model of $\mathcal{T}$. Suppose that $\mathcal{T}$ is satisfaction complete wrt. $\mathcal{L}$, and that P is executed on a CLP system for this language. Then:

1. If a goal G has a finite computation tree, with answers $c_1, \ldots, c_n$ then $P^*, \mathcal{T} \models G \leftrightarrow \exists_{-\mathcal{V}}(c_1 \vee \ldots \vee c_n)$, where $\mathcal{V}$ is the set of variables appearing in G.
2. If a goal G is finitely failed for P then $P^*, \mathcal{T} \models \neg G$.

The soundness of PropMiner is stated by the following theorem.

Theorem 2 (Soundness). The PropMiner algorithm produces propagation rules that are relevant wrt. $Base_{lhs}$ and valid wrt. $\mathcal{T}$ and P.

Proof. All C_{lhs} considered are interesting pattern wrt. $Base_{lhs}$, thus only relevant rules can be generated. If a rule of the form $C_{lhs} \Rightarrow false$ is produced then by property 2 in Theorem 1 this rule is valid. Suppose a rule of the form $C_{lhs} \Rightarrow C_{rhs}$ is generated. Then C_{rhs} is the lgg of a finite set of answers $\{Ans_1, \ldots, Ans_n\}$ obtained by the execution of the goal C_{lhs} on the program P. By property 1 in Theorem 1, we have $P^*, \mathcal{T} \models C_{lhs} \leftrightarrow \exists_{-\mathcal{V}}(Ans_1 \vee \ldots \vee Ans_n)$, where $\mathcal{V}$ is the set of variables appearing in C_{lhs}. Since C_{rhs} is the lgg of $\{Ans_1, \ldots, Ans_n\}$ then by [15] we know that $Ans_1 \vee \ldots \vee Ans_n \rightarrow C_{rhs}$. Thus $P^*, \mathcal{T} \models C_{lhs} \rightarrow \exists_{-\mathcal{V}}\ C_{rhs}$, i.e. $C_{lhs} \Rightarrow C_{rhs}$ is valid wrt. $\mathcal{T}$ and P. □

2.3 Interesting Rules for Constraint Solvers

The basic form of the PROPMINER algorithm given in Figure 2.2 produces a very large set of rules. Most of these rules are redundant (partly or completely) or propagates too weak constraints or on the contrary propagates too many stronger constraints (inflating considerably the constraint store at runtime) and thus may be of little interest to built a constraint solver.

We present in this section mandatory complementary processing that is integrated in the basic algorithm in order to generate rules of practical interest wrt. solver construction.

Consider again the CLP program of example 2. Let $Base_{lhs} = \{p(X,Y,Z)\}$ and let us use a richer set of constraints to form the lhs of the rules $Cand_{lhs} = \{X{\le}Z,\ Y{\le}X,\ X{=}Z, Y{=}Z,\ X{=}b,\ Y{=}a,\ Z{=}b\}$.

Among the rules generated by the basic algorithm PROPMINER, we have:

$$\{p(X,Y,Z)\} \Rightarrow \{X{\le}E,\ Y{=}E\}. \quad (1)$$
$$\{p(X,Y,Z),\ X{\le}Z\} \Rightarrow \{X{\le}a,\ Y{=}a,\ Z{\ne}b,\ X{\le}Z\}. \quad (2)$$
$$\{p(X,Y,Z),\ Y{\le}X\} \Rightarrow \{X{\le}E,\ Y{=}E,\ Y{\le}X\}. \quad (3)$$
$$\{p(X,Y,Z),\ X{=}Z\} \Rightarrow \{X{\le}a,\ Y{=}a,\ Z{\ne}b,\ X{=}Z\}. \quad (4)$$
$$\{p(X,Y,Z),\ Y{=}Z\} \Rightarrow \{X{\le}a,\ Y{=}a,\ Z{\ne}b,\ Y{=}Z\}. \quad (5)$$
$$\{p(X,Y,Z),\ X{=}b\} \Rightarrow \{X{\le}E,\ Y{=}E,\ X{=}b\}. \quad (6)$$
$$\{p(X,Y,Z),\ Y{=}a\} \Rightarrow \{X{\le}E,\ Y{=}E,\ Y{=}a\}. \quad (7)$$
$$\{p(X,Y,Z),\ Z{=}b\} \Rightarrow \{X{\le}W,\ Y{=}W,\ X{>}Z,\ Z{=}b\}. \quad (8)$$
$$\{p(X,Y,Z),\ X{\le}Z,\ Z{=}b\} \Rightarrow false. \quad (9)$$

Since the algorithm only imposes that the exploration ordering is a total ordering compatible with the subset ordering on the lhs, the real order of the rules generated may be slightly different according to implementation choices (see Section 2.4). However, the specific processing presented in this section can still be applied.

Removing redundancy The key idea of the simplification is to remove from the rhs of a rule R all constraints that can be derived from the lhs of R using the built-in solvers and the rules already generated. If the remaining rhs is empty then the whole rule can be suppressed.

For example, according to this process rule (6) is removed because its rhs is fully redundant wrt. its lhs and wrt. rule (1). For rule (2) only the rhs is modified and becomes $\{X{\le}a,\ Y{=}a,\ Z{\ne}b\}$, since $X{\le}Z$ is trivially entailed by the lhs of the rule.

Depending on the behavior of the built-in solvers, rule (4) may be only transformed into $\{p(X,Y,Z),\ X{=}Z\} \Rightarrow \{X{\le}a,\ Y{=}a,\ Z{\ne}b\}$ while if we know the semantics of $\le$ we may use rule (2) to derive the same constraints. If the built-in solver does not allow to discover this redundancy, then in our implementation (see Section 2.4) the user can add in a simple way propagation rules to derive explicitely logical consequences of the built-in constraints. In this example, one of the complementary rules that can be provided by the user is $\{X{=}Z\} \Rightarrow \{X{\le}Z\}$ which allows to find that rule (4) is then fully redundant wrt. rule (2).

This simplification process also applies to failure rules. Suppose that the built-in solver is able to detect that $Z{=}b \wedge Z{\neq}b$ is inconsistent, then the rule (9) is removed since it is redundant wrt. rule (2).

Generating stronger rhs If we consider rule (6) $\{p(X,Y,Z), X{=}b\} \Rightarrow \{X{\leq}E, Y{=}E, X{=}b\}$ the rhs constructed from the least general generalization of the answers obtained for the goal $\{p(X,Y,Z),\ X{=}b\}$ is in some sense too general. The execution of the goal gives two answers. One containing $\{Z{\neq}b\}$ and the other $\{X{>}Z,\ X{=}b\}$. From a semantical point of view, this leads clearly to $Z{\neq}b$ in both cases, but the least general generalization is mainly syntactical and do not retains this information.

If we want a richer rhs (containing $Z{\neq}b$) then we must have at hand a (built-in) solver that propagates $\{Z{\neq}b\}$ also in the second answer. If we do not have such a solver, then here again the user can provide himself complementary propagation rules (in this example the single rule $\{X{>}Y\} \Rightarrow \{X{\neq}Y\}$) to produce this piece of information.

Projecting variables For efficiency reasons in constraint solving it is particularly important to limit the number of variables.

Then a rule like $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E,\ Y{=}E\}$ should be avoided since it will create a new variable each time it is fired.

So, we simply project out such useless variables in the following way. We consider in turn each equality in the rhs of a rule. If this equality is of the form $E{=}F$ or $F{=}E$ where E and F are variables and E does not appear in the lhs of the rule, then we suppress this equality from the rhs and we apply the substitution transforming E into F to the whole remaining rhs.

More subtle situations may arise. Suppose that the second clause of the program given in example 2 was $p(X,Y,Z) \leftarrow X{\leq}W \wedge Y{=}W \wedge Z{\neq}a$. Then, the first rule generated would have been $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}E,\ Y{=}E,\ Z{\neq}F\}$. And then projecting out E would transform it into $\{p(X,Y,Z)\} \Rightarrow \{X{\leq}Y,\ Z{\neq}F\}$. Then, during constraint solving the application of this rule will add to the store the constraint $Z{\neq}F$, where F is a new variable. This phenomena leads in general to a rather inefficient solving process. So, we propose the following optional treatment: When all other previous processing has been performed (simplification, additional propagation and projection of variable in equalities) the user can choose to apply a strict *range restriction* criteria: all constraints in the rhs containing a variable that does not appear in the lhs is removed (e.g., $Z{\neq}F$ in the previous rule). This range restriction criteria is applied in all examples presented in this paper. However, it should be noticed that this process remains optional since this simplification criteria is purely syntactic and does not guarantee that the constraints removed from the rhs are semantically redundant, and thus may produce weaker rules (although still valid).

2.4 Implementation Issues

The key aspects of our implementation of the PropMiner algorithm are presented in this section. The prototype has been developed under SICStus Prolog 3.7.1. It is written in Prolog and takes advantage of the rule-based programming language Constraint Handling Rules (CHR) [6] supported in this environment.

Using CHR. The CHR language facilitates in two ways the implementation of the important processing described in Section 2.3. Firstly, we can use the rules generated as CHR rules and then run CHR to decide if a rule propagates new constraints wrt. the rules we have already. Secondly, the user can directly add new rules to perform complementary propagations wrt. the built-in solvers as mentioned in Section 2.3.

Clause encoding. It should be noticed that in this environment the equality $=$ is reserved to specify unification. So in practice, we use another binary predicate to denote the equality constraint. Moreover, the bindings of the variables due to the resolution steps are not handled explicitely as equalities in the store. Suppose that the third clause of the program given in example 2 was written under the form $q(X, a, Z) \leftarrow X{\leq}a \wedge Z{\neq}b$. Then, for the goal $\{p(X, Y, Z),\ X{\leq}Z\}$ we may have not collected the constraint $Y{=}a$ explicitely and thus $Y{=}a$ will not appear in the rhs of rule (2). Thus, we simply preprocess the clauses so that the atom in the head of a clause does not contain functors (including constants) and coreferences. The corresponding functors and coreferences are simply encoded by equality constraints in the body of the clause. For example a head of the form $p(X, a, X)$ will be transformed into $p(X, Y, Z)$ and $X{=}Z \wedge Y{=}a$ will be added to the body.

Enumeration of lhs. The PropMiner algorithm enumerates the possible lhs (the elements in L). The implementation of this enumeration is based on the exploration of a tree corresponding to the lhs search space. This tree is explored using a depth first strategy. As in [3], the branches are expanded using a partial ordering on the lhs candidates such that the more general lhs are examined before more specialized ones. The partial ordering used in our implementation is the θ-subsumption ordering [15].

3 Practical Uses of PropMiner

In this section, we show on examples that a practical application of our approach lies in solver development. All the set of rules presented in this section have been generated in a few seconds on a PC Pentium 3 with 128 MBytes of memory and a 500 MHZ processor.

For convenience, we introduce the following notation. Let c be a constraint symbol of arity 2 and D_1 and D_2 be two sets of terms. We define $atomic(c, D_1, D_2)$ as the set of all constraints built from c over $D_1 \times D_2$. More precisely,
$atomic(c, D_1, D_2) = \{c(\alpha, \beta) \mid \alpha \in D_1 \text{ and } \beta \in D_2\}$.

Example 3. For the minimum predicate $min(A,B,C)$ defined by the CLP program of Example 1, the PROPMINER algorithm with the following input

$$\begin{aligned} Base_{lhs} &= \{min(A,B,C)\} \\ Cand_{lhs} &= atomic(=,\{A,B,C\},\{A,B,C\}) \,\cup \\ &\quad\; atomic(\neq,\{A,B,C\},\{A,B,C\}) \,\cup \\ &\quad\; atomic(\leq,\{A,B,C\},\{A,B,C\}) \end{aligned}$$

generates the 6 propagation rules presented in Example 1.
It should be noticed that to be able to generate the first rule, the following rules for equality and less-or-equal constraints have to be present in the built-in solver to ensure the generation of stronger rhs (as illustrated in Section 2.3):

$$\begin{aligned} X{\leq}Y \;\wedge\; Y{\leq}Z \;&\Rightarrow\; X{\leq}Z. \\ X{=}Y \;&\Rightarrow\; X{\leq}Y. \end{aligned}$$

If these rules are not already in the built-in solver, in our implementation the user can provide them very easily by means of CHR rules (see Section 2.4). Moreover, using this possibility, PROPMINER can incorporate additional knowledge given by the user about the predicate of interest. For example, the user can express the symmetry of *min* with respect to the the first and second arguments by the rule:

$$min(A,B,C) \;\Rightarrow\; min(B,A,C).$$

If this rule is provided by the user as a CHR rule, it completes the built-in solver and then the PROPMINER algorithm generates only the following simplified set of 4 rules:

$$\begin{aligned} min(A,B,C) \;&\Rightarrow\; C{\leq}A \;\wedge\; C{\leq}B. \\ min(A,B,C) \;\wedge\; A{=}B \;&\Rightarrow\; A{=}C. \\ min(A,B,C) \;\wedge\; C{\neq}B \;&\Rightarrow\; C{=}A. \\ min(A,B,C) \;\wedge\; B{\leq}A \;&\Rightarrow\; C{=}B. \end{aligned}$$

Example 4. If we consider the maximum predicate *max*, a set of rules similar to the rules for *min* is generated by PROPMINER. Then the user has the possibility to add these two sets of rules to the built-in solver and to execute PROPMINER to generate interaction rules between *min* and *max*. This execution is performed with the following input

$$\begin{aligned} Base_{lhs} &= \{min(A,B,C) \wedge max(D,E,F)\} \\ Cand_{lhs} &= atomic(\neq,\{A,B,C\},\{D,E,F\}) \end{aligned}$$

and a CLP program consisting of the definitions of *min* and *max*. Since the propagation rules specific to *min* and *max* alone have been added to the built-in solver, PropMiner takes advantage of these rules to simplify many redundancies. Thus only 10 propagation rules specific to the conjunction of min with max are generated. Examples of rules are:

$$min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ C{\neq}E \ \wedge \ C{\neq}D \ \Rightarrow \ F{\neq}C.$$
$$min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ B{\neq}D \ \wedge \ A{\neq}D \ \Rightarrow \ D{\neq}C.$$
$$min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ C{\neq}E \ \wedge \ B{\neq}D \ \wedge \ A{\neq}F \ \Rightarrow \ F{\neq}C.$$
$$min(A,B,C) \ \wedge \ max(D,E,F) \ \wedge \ C{\neq}D \ \wedge \ B{\neq}F \ \wedge \ A{\neq}E \ \Rightarrow \ F{\neq}C.$$

4 Handling Recursive Constraint Definitions

In this section, we show informally that the algorithm PropMiner can be applied when the CLP program P defining the constraint predicates is recursive and may lead to non-terminating executions.
As presented in Figure 2.2, for each possible rule lhs in L (denoted by C_{lhs}) the algorithm needs to collect in finite time all answers to the goal C_{lhs} wrt. the program P. In general, we cannot guarantee such a termination property, but we can use standard Logic Programming solutions developed to handle recursive clauses. For example, we can prefer a resolution based on the OLDT [19] scheme that ensures finite refutations more often than a resolution following the SLD principle (e.g., with the OLDT resolution the execution always terminates for Datalog programs).
We can also decide to bound the depth of the resolution to stop the execution of a goal that may cause non-termination. In this case, if the execution of goal C_{lhs} has a resolution depth exceeding a given threshold, we interrupt this execution and proceed with the next possible lhs in L. Of course this strategy may be too restrictive, in the sense that it may stop too early some terminating executions and thus may avoid the generation of some interesting rules.

Example 5. Consider the well-known ternary *append* predicate for lists, which holds if its third argument is a concatenation of the first and the second argument. It is usually implemented by these two clauses:

$$append(X,Y,Z) \ \leftarrow \ X{=}[] \ \wedge \ Y{=}Z.$$
$$append(X,Y,Z) \ \leftarrow \ X{=}[H|X1] \ \wedge \ Z{=}[H|Z1] \ \wedge \ append(X1,Y,Z1).$$

Then, if we bound the resolution depth to discard non-terminating executions, the algorithm PropMiner terminates and using the appropriate input produces, among others, the following rules:

$$append(A,B,C) \ \wedge \ A{=}B \ \wedge \ C{=}[D] \ \Rightarrow \ false.$$
$$append(A,B,C) \ \wedge \ B{=}C \ \wedge \ C{=}[D] \ \Rightarrow \ A{=}[].$$
$$append(A,B,C) \ \wedge \ C{=}[] \ \Rightarrow \ B{=}[] \ \wedge \ A{=}[].$$
$$append(A,B,C) \ \wedge \ A{=}[] \ \Rightarrow \ B{=}C.$$

5 Conclusion and Related Work

We have presented an approach to generate rule-based constraint solvers from the intentional definition of the constraint predicates given by means of a CLP program. The generation is performed in two steps. In a first step, it produces propagation rules using the algorithm PROPMINER described in Section 2, and in a second step it transforms some of these rules into simplification rules using the method proposed in [4].

Now, we briefly compare our work to other approaches and give directions for future work.

- In [5, 17, 3] first steps towards automatic generation of propagation rules have been done. In these approaches the constraints are defined extensionally over finite domains by e.g. a truth table or their solution tuples. Thus, this paper can be seen as an extension of these previous works towards constraints defined intensionally over infinite domains. Over finite domains the algorithm PROPMINER, can be used to generate the rules produces by the other methods.

  Example 6. For the boolean negation $neg(X, Y)$, the algorithm PROPMINER and the algorithm described in [3] generate the same rules:

$$\begin{aligned} neg(X, X) &\Rightarrow false. \\ neg(X, 1) &\Rightarrow X{=}0. \\ neg(X, 0) &\Rightarrow X{=}1. \\ neg(1, Y) &\Rightarrow Y{=}0. \\ neg(0, Y) &\Rightarrow Y{=}1. \end{aligned}$$

- *Generalized Constraint Propagation* [16] extends the propagation mechanism from finite domains to arbitrary domains. The idea is to find and propagate a simple *approximation* constraint that is a kind of least upper bound of a set of computed answers to a goal. In contrast to our approach where the generation of rules is done once at compile time, generalized propagation is performed at runtime.
- *Constructive Disjunction* [8, 20] is a way to extract common information from disjunctions of constraints over finite domains. We are currently investigating how constructive disjunction can be used in our case to enhance the computation of the least upper bound of set of answers in the case of constraints over finite domains. One advantage is that this approach can collect more information since it takes into account the semantics of the arithmetic operators, comparison predicates, and interval constraints.
- In ILP [12] and ICLP [13, 11, 10, 18], the user is interested to find out logic programs and CLP programs from examples. In our case, we generate constraint solvers in the form of propagation and simplification rules, using the

definition of the constraint predicates given by means of a CLP program. We used techniques also used in ILP and ICLP (e.g., [15]), and it is important to consider which of the works done in these fields may be used for the generation of constraint solvers.

To our knowledge, the work done on Generalized Constraint Propagation, Constructive Disjunction, and in the fields of ILP and ICLP have not previously been adapted or applied to the generation of rule-based constraint solvers.

Future work includes the extension of the algorithm PropMiner to generate more information to be propagated in the right hand side of the rules. In the current algorithm, the computation of the least upper bound of set of answers is based on [15] which does not rely on the semantics of the constraints in the answers. As illustrated in Section 2.3 and Section 3, the user can provide by hand propagation rules to take into account (partially) this semantics, but, as it has been pointed out to us, approaches like [14] can be used to embed this semantics in a more general way and directly in the computation of the least upper bound. Another complementary aspect that needs to be investigated is the completeness of the solvers generated. It is clear that in general this property cannot be guaranteed, but in some cases it may be possible to check it, or at least to characterize the kind of consistency the solver can ensure.

Acknowledgments

We would like to thank Thom Frühwirth for helpful discussions. We also grateful to the anonymous referees for many helpful suggestions which undoubtedly improved the paper.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of the third International Conference on Principles and Practice of Constraint Programming, CP'97*, LNCS 1330, pages 252–266. Springer-Verlag, November 1997.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the Second International Conference on Principles and Practice of Constraint Programming*, 4(2):133–165, May 1999.
3. S. Abdennadher and C. Rigotti. Automatic generation of propagation rules for finite domains. In *Proc. of the 6th International Conference on Principles and Practice of Constraint Programming, CP'00*, LNCS 1894, pages 18–34. Springer-Verlag, September 2000.
4. S. Abdennadher and C. Rigotti. Using confluence to generate rule-based constraint solvers. In *Proc. of the third International Conference on Principles and Practice of Declarative Programming.* ACM Press, September 2001. To appear.

5. K. Apt and E. Monfroy. Automatic generation of constraint propagation algorithms for small finite domains. In *Proc. of the 5th International Conference on Principles and Practice of Constraint Programming, CP'99*, LNCS 1713, pages 58–72. Springer-Verlag, October 1999.
6. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.
7. T. Frühwirth. Proving termination of constraint solver programs. In *New Trends in Constraints*, pages 298–317. LNAI 1865, 2000.
8. P. V. Hentenryck, V. Saraswat, and Y. Deville. Desing, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):139–164, 1998.
9. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19-20:503–581, 1994.
10. L. Martin and C. Vrain. Induction of constraint logic programs. In *Proc. of the International Conference on Algorithms and Learning Theory*, LNCS 1160, pages 169–176. Springer-Verlag, October 1996.
11. F. Mizoguchi and H. Ohwada. Constrained relative least general generalization for inducing constraint logic programs. *New Generation Computing*, 13:335–368, 1995.
12. S. Muggleton and L. De Raedt. Inductive Logic Programming : theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
13. S. Padmanabhuni and A. K. Ghose. Inductive constraint logic programming: An overview. In *Learning and reasoning with complex representations*, LNCS 1359, pages 1–8. Springer-Verlag, 1998.
14. C. Page and A. Frisch. Generalization and learnability: a study of constrained atoms. In *Inductive Logic Programming*, pages 29–61. London: Academic Press, 1992.
15. G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
16. T. L. Provost and M. Wallace. Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16(3):319–359, 1993.
17. C. Ringeissen and E. Monfroy. Generating propagation rules for finite domains: A mixed approach. In *New Trends in Constraints*, pages 150–172. LNAI 1865, 2000.
18. M. Sebag and C. Rouveirol. Constraint inductive logic programming. In *Advances in ILP*, pages 277–294. IOS Press, 1996.
19. H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proc. of the 3rd International Conference on Logic Programming*, LNCS 225, pages 84–98. Springer-Verlag, 1986.
20. J. Würtz and T. Müller. Constructive disjunction revisited. In *Proc. of the 20th German Annual Conference on Artificial Intelligence*, LNAI 1137, pages 377–386. Springer-Verlag, 1996.