Constructing Rule-based Solvers for Intentionally-defined Constraints

Ingi Sobhi[1], Slim Abdennadher[1], and Hariolf Betz[2]

[1] Faculty of Media Engineering and Technology, German University in Cairo, Egypt
[Slim.Abdennadher,Ingi.Sobhi]@guc.edu.eg
[2] Faculty of Engineering and Computer Science, University of Ulm, Germany
Hariolf.Betz@uni-ulm.de

Abstract. Developing constraint solvers which are key requisites of constraint programming languages is time consuming and difficult. In this paper, we propose a generic algorithm that symbolically constructs rule-based solvers from the intensional definition of the constraint. Unlike the well-established "generate and test" approach, our symbolic construction approach is capable of generating recursive rules from a recursive constraint definition. Combining the two approaches gives better filtering capabilities than either of the approaches acting alone.

1 Introduction

> "Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." [E. Freuder]

The validity of this statement for a Constraint Logic Programming (CLP) language is contingent on the existence of constraint solvers. These associate constraints with filtering algorithms that remove variable values which cannot belong to any solution of the problem.

Constraint Handling Rules (CHR) is a multi-headed guarded and concurrent constraint logic programming language. To incorporate constraint solvers in CHR, a scheme was proposed in [1] to automatically derive the solvers given the intentional definition of the constraints. The scheme is based on a generate and test approach where rule candidates are enumerated and tested for validity against the constraint definition. Although the approach performs an extensive search for valid rules, given a recursive constraint definition it is unable to generate recursive rules.

To overcome this, we propose a scheme where valid rules are symbolically constructed from the clauses of a CLP program defining the constraint. The idea behind the construction stems from the observation that if in a non-overlapping CLP program the execution of a clause leads to a solution, the execution of all other clauses will not. Thus our constructed CHR rules simplify the constraint to the body of a clause only if all other clauses do not hold. Moreover, we combine the two schemes to achieve better filtering.

Example 1 (Motivation). Consider the lexicographic order constraint [2–4]. Given two sequences L_1 and L_2 of variables of the same length, then lex holds if L_1 is lexicographically smaller than or equal to L_2. The following CLP program defines the $lex(L_1, L_2)$ constraint:

$$\begin{aligned} &lex([],[]) \\ &lex([X_1|T_1],[X_2|T_2]) \;\leftarrow\; X_1{<}X_2 \\ &lex([X_3|T_3],[X_4|T_4]) \;\leftarrow\; X_3{=}X_4 \wedge lex(T_3,T_4) \end{aligned}$$

The generate and test approach [1] generates rules that reason about the first elements of the two lists such as:

$$lex(L_1, L_2) \Rightarrow L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \mid X_1{\leq}X_2 \tag{1}$$

The symbolic construction approach proposed in this paper generates the following solver:

$$lex(L_1, L_2) \Leftrightarrow L_1{=}[] \vee L_2{=}[] \mid L_1{=}[] \wedge L_2{=}[] \tag{2}$$

$$lex(L_1, L_2) \Leftrightarrow L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\neq}X_2 \mid X_1{<}X_2 \tag{3}$$

$$lex(L_1, L_2) \Leftrightarrow L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2 \mid X_1{=}X_2 \wedge lex(T_1, T_2) \tag{4}$$

Given the query $\langle lex([A_1, A_2, A_3], [B_1, B_2, B_3])\rangle$ where the domains of the variables are defined as follows:

$$\begin{gathered} A_1{=}\{1,3,4\},\ A_2{=}\{2,3,4\},\ A_3{=}\{1,2\} \\ B_1{=}\{1\},\quad B_2{=}\{2\},\quad B_3{=}\{0,1,2\} \end{gathered}$$

The generate and test approach enforces the constraint $A_1{\leq}B_1$, which removes the values $\{3,4\}$ from the domain of A_1. The solution becomes:

$$\begin{gathered} A_1{=}\{1\},\ A_2{=}\{2,3,4\},\ A_3{=}\{1,2\}, \\ B_1{=}\{1\},\quad B_2{=}\{2\},\quad B_3{=}\{0,1,2\}, \\ lex([A_1, A_2, A_3], [B_1, B_2, B_3]) \end{gathered}$$

For the symbolic construction approach since $A_1{\geq}B_1$, rule (4) is executed enforcing equality on the values of A_1 and B_1 before calling lex recursively on the remaining list elements. Since $A_2{\geq}B_2$ rule (4) is applied again, whereas for A_3 and B_3 no rule is applicable. The solution becomes:

$$\begin{gathered} A_1{=}\{1\},\ A_2{=}\{2\},\ A_3{=}\{1,2\}, \\ B_1{=}\{1\},\ B_2{=}\{2\},\ B_3{=}\{0,1,2\}, \\ lex([A_3], [B_3]) \end{gathered}$$

Combining both approaches prunes the domains of the variables further since rule (1) is applicable to $lex([A_3], [B_3])$ and filters the domain of B_3. The combined solution becomes:

$$\begin{gathered} A_1{=}\{1\},\ A_2{=}\{2\},\ A_3{=}\{1,2\}, \\ B_1{=}\{1\},\ B_2{=}\{2\},\ B_3{=}\{1,2\}, \\ A_3{\leq}B_3, lex([A_3], [B_3]) \end{gathered}$$

We will proceed with the lex constraint in all examples of this paper.

The paper is a revised and extended version of [5] and is organized as follows. In section 3, we present the symbolic construction approach and prove soundness and termination of the constructed solvers. In section 4, we apply post-processing methods to improve the run-time complexity of the solvers. Finally, section 5 combines the symbolic construction approach with the "generate and test" approach to achieve better filtering.

2 Preliminaries

2.1 Intentional Definition

Let p be a constraint. A CLP program P defines p if p occurs with the same arity in the head of all the clauses and all true instances of p are accounted for (closed world assumption). The program P is of the usual form:

$$p(\bar{t}_1) \leftarrow C_1, p(\bar{t}_2) \leftarrow C_2, \ldots, p(\bar{t}_n) \leftarrow C_n$$

where $\bar{t}_i$ stands for a sequence of terms and C_i is a conjunction of built-in and user-defined constraints. Built-in constraints are those defined by a constraint theory and for which solvers are available. These solvers are assumed to be well-behaved (terminating and confluent), closed under negation, and achieve arc-consistency. User-defined constraints are those for which solvers are needed. The symbolic construction approach requires that there are no variables in C_i that are not in $\bar{t}_i$ and that all clauses are non-overlapping (i.e. in a computation at most one clause can lead to a solution).

Definition 1. *The logical reading of P denoted by P^* is given by its* Clark completion *[6]:*

$$\forall \bar{x}\ (p(\bar{x}) \leftrightarrow \bigvee_{i=1}^{n} \exists \bar{y}_i\ (\bar{x}{=}\bar{t}_i \wedge C_i))$$

where $\bar{x}$ is a sequence of distinct fresh variables and $\bar{y}_i$ is the sequence of variables in $\bar{t}_i$. The expression $\bar{x}{=}\bar{t}_i$ stands for the conjunction of equations between respective elements of the sequences $\bar{x}$ and $\bar{t}_i$.

Example 2 (Clark Completion). The Clark completion of the CLP program defining *lex* is:

$$\begin{aligned}
\forall L_1, L_2\ & lex(L_1, L_2)\ \leftrightarrow \\
& (L_1{=}[] \wedge L_2{=}[]) \vee \\
& \exists X_1, X_2, T_1, T_2\ (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{<}X_2) \vee \\
& \exists X_3, X_4, T_3, T_4\ (L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{=}X_4 \wedge lex(T_3, T_4))
\end{aligned}$$

2.2 Constraint Solver

CHR [7] specifies how new constraints interact with the constraint store and is thus especially suited for writing constraint solvers. It has two main rule types:

$$\begin{aligned} &\text{Simplification Rule:} && H \Leftrightarrow G \mid B \\ &\text{Propagation Rule:} && H \Rightarrow G \mid B \end{aligned}$$

where the *head* H are user-defined constraints, the *guard* G are built-in constraints and the *body* constraints B are both.

Definition 2. *The* logical meaning *of a simplification rule is a logical equivalence provided the guard holds:*

$$\forall \bar{x} \forall \bar{y}\ (G \rightarrow (H \leftrightarrow \exists \bar{z}\ B))$$

where $\bar{x}$ is the set of variables occurring in H, the variables $\bar{y}$ are the set occurring in G but not in H and $\bar{z}$ are the variables occurring in B only. Similarly, the logical meaning of a propagation rule is an implication provided the guard holds.

Prompted with a query, applicable rules are executed until a fixpoint is reached where no more rules can be applied or a contradiction occurs. A rule is applicable provided that constraints from the query match the head and imply the guard. Execution of a simplification rule rewrites constraints that match the head by the body while execution of a propagation rule adds the body constraints to the constraint store.

2.3 Generate and Test Approach

In this section, we summarize the generate and test approach presented in [1]. Given a CLP program defining the constraint as well as the syntactic form of the candidate rules defined by the following sets:

- $Base_{lhs}$ contains constraints that must appear in the head of all rules,
- $Cand_{lhs}$ contains constraints to be used in conjunction with $Base_{lhs}$ to form the head, and
- $Cand_{rhs}$ contains constraints that may appear in the body.

The generate and test approach generates valid rules as follows. Candidate propagation rules of the form $H \Rightarrow B$ are enumerated, and subjected to a validity test based on the observation that a rule is valid if the execution of the goal $H \wedge \neg(B)$ finitely fails with respect to the CLP program.

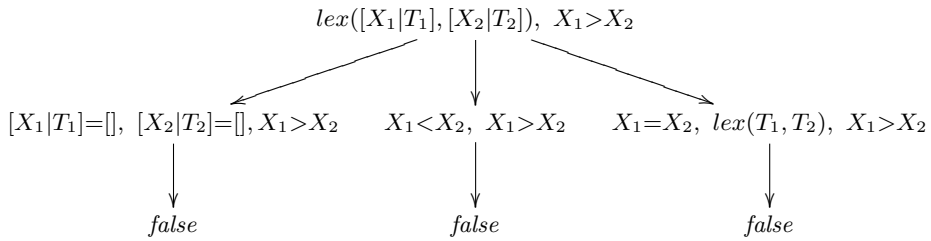
Example 3 (Generate and Test Approach). Given the syntactic form of candidate rules for lex as:

$Base_{lhs}$=$\{lex(L_1, L_2)\}$
$Cand_{lhs}$=$\{L_1$=$[], L_2$=$[], L_1$=$[X_1|T_1], L_2$=$[X_2|T_2], X_1 \leq X_2, X_1 > X_2, X_1 < X_2, \ldots\}$
$Cand_{rhs}$=$Cand_{lhs}$

The generate and test approach generates (among others) the following rule for lex:

$$lex([X_1|T_1],[X_2|T_2]) \Rightarrow X_1 \leq X_2$$

The rule is generated since executing the goal $\langle lex([X_1|T_1],[X_2|T_2]), X_1 > X_2\rangle$ fails as demonstrated by the following derivation tree:



3 Symbolic Construction Approach

The symbolic construction approach (Fig. 1) constructs a solver for a constraint by symbolically transforming the Clark completion of the CLP program defining the constraint to semantically valid rules. The idea of the transformation stems from the observation that in a non-overlapping CLP program if the execution of one clause leads to a solution then the execution of all other clauses will not. Thus to construct a rule that simplifies the constraint to the body of one clause, the negation of the bodies of all other clauses is added to the guard. This ensures that the rule is applicable only when all other clauses are not valid and hence maintains consistency with the constraint definition.

---

begin
  p: left hand side of the Clark completion
  $Disjuncts$: set of disjuncts of right hand side of the Clark completion
  $Rules$={}: resultant rule set

  for each D **in** $Disjuncts$ **do**
    $Other$=$Disjuncts\backslash\{D\}$
    $Rules$=$Rules \cup \{p \Leftrightarrow \neg Other \mid D\}$
  end for
end

Fig. 1. Symbolic Construction Algorithm

3.1 Guard Determination

More formally, given the definition of a constraint $p(\bar{x})$:

$$\forall \bar{x} \left(p(\bar{x}) \leftrightarrow \bigvee_{i=1}^{n} \exists \bar{y}_i \, (\bar{x}{=}\bar{t}_i \wedge C_i) \right)$$

The symbolic construction algorithm constructs rules of the form:

$$p(\bar{x}) \Leftrightarrow \neg \bigvee_{j=1, j\neq i}^{n} \exists \bar{y}_j \, (\bar{x}{=}\bar{t}_j \wedge C_j) \mid \bar{x}{=}\bar{t}_i \wedge C_i \qquad \text{for each } i \in \{1, \ldots, n\}$$

where $\bar{x}{=}\bar{t}_j$ stands for the conjunction of equations between respective elements of the sequences $\bar{x}$ and $\bar{t}_j$. According to the soundness proof (given in the next section), this is equivalent to:

$$p(\bar{x}) \Leftrightarrow \bigwedge_{j=1, j\neq i}^{n} \forall \bar{y}_j \, (\bar{x}{\neq}\bar{t}_j) \vee \exists \bar{y}_j \, (\bar{x}{=}\bar{t}_j \wedge \neg C_j) \mid \bar{x}{=}\bar{t}_i \wedge C_i$$

where $\bar{x}{\neq}\bar{t}_j$ stands for the disjunction of negated equations between respective elements of the sequences $\bar{x}$ and $\bar{t}_j$ and $\neg C_j$ is a disjunction of negated constraints. The symbolic construction approach distinguishes between the two cases for negated constraints: Negated built-ins are replaced by the corresponding positive constraints since built-ins are closed under negation. Negated user-defined constraints are discarded and constructing an entailment checker that determines if a user-defined constraint does not hold is left for future work. Thus, the general form of the rules is:

$$p(\bar{x}) \Leftrightarrow \bigwedge_{j=1, j\neq i}^{n} \bigvee_{k=1}^{|x|+m_j} E_j^k \mid \bar{x}{=}\bar{t}_i \wedge C_i$$

where

$$E_j^k {=} \forall \bar{y}_j^k \left(x^k {\neq} t_j^k \right) \qquad \text{for } k \in \{1, \ldots, |x|\},$$

$$E_j^{|x|+k} {=} \exists \bar{y}_j \left(\bar{x}{=}\bar{t}_j \wedge \neg c_j^k \right) \qquad \text{for } k \in \{1, \ldots, m_j\},$$

the $\bar{y}_j^k$ is the sequence of variables in the term t_j^k and m_j is the number of built-in constraints in C_j.

Example 4 (Symbolic Construction Approach). The *lex* constraint has three disjuncts namely:

$$\begin{array}{ll} D_1: & L_1{=}[] \wedge L_2{=}[] \\ D_2: & \exists X_1, X_2, T_1, T_2 \; (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{<}X_2) \\ D_3: & \exists X_3, X_4, T_3, T_4 \; (L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{=}X_4 \wedge lex(T_3, T_4)) \end{array}$$

To construct a rule that simplifies *lex* to the first disjunct, the negation of the other two disjuncts is added to the guard:

$$\neg(\exists X_1, X_2, T_1, T_2\ (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{<}X_2) \vee \\ \exists X_3, X_4, T_3, T_4\ (L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{=}X_4 \wedge lex(T_3, T_4)))$$

This is equivalent to:

$$(\forall X_1, T_1(L_1{\neq}[X_1|T_1])\ \vee\ \forall X_2, T_2(L_2{\neq}[X_2|T_2]) \vee \\ \exists X_1, X_2, T_1, T_2(L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2)) \wedge \\ (\forall X_3, T_3(L_1{\neq}[X_3|T_3])\ \vee\ \forall X_4, T_4(L_2{\neq}[X_4|T_4]) \vee \\ \exists X_3, X_4, T_3, T_4(L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4))$$

where negated built-ins are replaced by the corresponding positive constraints and negated user-defined constraints discarded.

Since the arguments of *lex* are (ordered) lists, the constraint $\forall X, T\ L{\neq}[X|T]$ can be simplified to $L{=}[]$. The expression becomes:

$$(L_1{=}[]\ \vee\ L_2{=}[]\ \vee\ (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2)) \wedge \\ (L_1{=}[]\ \vee\ L_2{=}[]\ \vee\ (L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4))$$

Thus the constructed rule is:

$$lex(L_1, L_2) \Leftrightarrow \\ (L_1{=}[]\ \vee\ L_2{=}[]\ \vee\ (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2)) \wedge \\ (L_1{=}[]\ \vee\ L_2{=}[]\ \vee\ (L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4)) \\ \mid L_1{=}[] \wedge L_2{=}[]$$

3.2 The Solver Properties

In this section we prove soundness and termination as well as discuss completeness of the constructed solvers.

Soundness. A simplification rule $H \Leftrightarrow G \mid B$ is *valid* w.r.t. a CLP program P and the constraint theory CT iff $P^* \cup CT \models \forall \bar{x}\ (\exists \bar{y}\ G \rightarrow (H \leftrightarrow \exists \bar{z}\ B))$.

Theorem 1 (Soundness). *The symbolic construction algorithm constructs valid simplification rules w.r.t. the CLP program and the constraint theory.*

Proof (Soundness). Consider Clark's completion of the CLP program defining a constraint p:

$$\forall \bar{x} \left(p(\bar{x}) \leftrightarrow \left(\bigvee_{i=1}^{n} \exists \bar{y}_i\ \ (\bar{x}{=}\bar{t}_i \wedge C_i) \right) \right)$$

For every $i \in \{1, \ldots, n\}$, we therefore have:

$$\forall \bar{x} \left(\left(\neg \bigvee_{j=1, j \neq i}^{n} \exists \bar{y}_j \, (\bar{x}{=}\bar{t}_j \wedge C_j) \right) \rightarrow (p(\bar{x}) \leftrightarrow \exists \bar{y}_i \;\; (\bar{x}{=}\bar{t}_i \wedge C_i)) \right)$$

and consequently:

$$\forall \bar{x} \left(\left(\bigwedge_{j=1, j \neq i}^{n} \neg \exists \bar{y}_j \, (\bar{x}{=}\bar{t}_j \wedge C_j) \right) \rightarrow (p(\bar{x}) \leftrightarrow \exists \bar{y}_i \;\; (\bar{x}{=}\bar{t}_i \wedge C_i)) \right)$$

which is the logical reading of a CHR rule:

$$p(\bar{x}) \Leftrightarrow G \mid \bar{x}{=}\bar{t}_i \wedge C_i$$

where G is equivalent to:

$$\bigwedge_{j=1, j \neq i}^{n} \neg \exists \bar{y}_j \, (\bar{x}{=}\bar{t}_j \wedge C_j)$$

Recall that $\bar{y}_j$ denotes the variables in $\bar{t}_j$ and that $\bar{y}_j$ is disjoint from $\bar{x}$. Therefore, for every $j \in \{1, \ldots, n\}$ and every valuation of $\bar{x}$ such that $\bar{x} = \bar{t}_j$ is satisfiable, there exists a sequence of terms $\bar{u}_j$ such that:

$$(\bar{x} = \bar{t}_j) \Leftrightarrow (\bar{y}_j = \bar{u}_j)$$

This observation guarantees the existence of a function u_j for each $j \in \{1, \ldots, n\}$ that maps from sequences of terms to sequences of terms such that:

$$(\exists \bar{y}_j \;\; \bar{x}{=}\bar{t}_j) \Rightarrow ((\bar{x}{=}\bar{t}_j) \Leftrightarrow (\bar{y}_j{=}u_j(\bar{x})))$$

and consequently:

$$(\bar{x} = \bar{t}_j) \Rightarrow (\bar{y}_j{=}u_j(\bar{x}))$$

Using function u_j, we have that:

$$\neg \exists \bar{y}_j \;\; (\bar{x}{=}\bar{t}_j \wedge C_j)$$

is equivalent to:

$$\neg \exists \bar{y}_j \;\; (\bar{x}{=}\bar{t}_j \wedge \bar{y}_j{=}u_j(\bar{x}) \wedge C_j)$$

From there, the substitution property of equality gives us:

$$\neg \exists \bar{y}_j \;\; (\bar{x}{=}\bar{t}_j \wedge C_j[\bar{y}_j / u_j(\bar{x})])$$

We move the negation to the inside of the formula and get:

$$\forall \bar{y}_j \;\; (\bar{x}{\neq}\bar{t}_j \vee \neg C_j[\bar{y}_j / u_j(\bar{x})])$$

As the variables $\bar{y}_j$ do not appear in the formula $\neg C_j[\bar{y}_j/u_j(\bar{x})]$, we can move it outside of the universal quantification:

$$\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee \neg C_j[\bar{y}_j/u_j(\bar{x})]$$

Applying $(A \vee B) \Leftrightarrow (A \vee (\neg A \wedge B))$ gives us:

$$\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee (\exists \bar{y}_j \ (\bar{x} = \bar{t}_j) \wedge \neg C_j[\bar{y}_j/u_j(\bar{x})])$$

As the variables $\bar{y}_j$ do not appear in the formula $\neg C_j[\bar{y}_j/u_j(\bar{x})]$, we can move it into the scope of their existential quantification:

$$\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee \exists y_j \ (\bar{x} = \bar{t}_j \wedge \neg C_j[\bar{y}_j/u_j(\bar{x})])$$

According to the definition of the function u_j, $\bar{x} = \bar{t}_j$ implies $\bar{y}_j = u_j(\bar{x})$:

$$\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee \exists y_j \ (\bar{x} = \bar{t}_j \wedge \bar{y}_j = u_j(\bar{x}) \wedge \neg C_j[\bar{y}_j/u_j(\bar{x})])$$

We apply the substitution property of equality again to get:

$$\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee \exists y_j \ (\bar{x} = \bar{t}_j \wedge \bar{y}_j = u_j(\bar{x}) \wedge \neg C_j)$$

As $(\bar{x} = \bar{t}_j) \Rightarrow (\bar{y}_j = u_j(\bar{x}))$, this is equivalent to:

$$\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee \exists y_j \ (\bar{x} = \bar{t}_j \wedge \neg C_j)$$

Therefore, the guard G of the generated CHR rule is equivalent to:

$$\left(\bigwedge_{j=1, j \neq i}^{n} (\forall \bar{y}_j \ \ (\bar{x} \neq \bar{t}_j) \vee \exists y_j \ (\bar{x} = \bar{t}_j \wedge \neg C_j)) \right)$$

Termination. In [8] proving the termination of CHR solvers is based on polynomial interpretations where the rank of a term or an atom is defined by a linear positive combination of the rankings of its arguments. The basic idea is to prove that the rank of the head of a rule is strictly larger than that of its body. Moreover, built-in solvers are assumed to be well-behaved (terminating and confluent) and thus the rank of built-in constraints is defined as 0.

Theorem 2 (Termination). *For constraints defined by a CLP program where the rank of the head of every clause is strictly larger than that of its body, the symbolic construction approach constructs terminating solvers.*

Proof (Termination). If for each clause of the CLP program, the rank of the head of the clause is strictly larger than that of its body, then the constructed solver terminates. The head and body of a constructed rule are the same as the clause and only built-in constraints which are defined as 0 are added to the guard. Thus the head of a constructed rule is strictly larger than that of its body and the solver terminates.

Completeness. The constructed solvers can not guarantee propagation completeness for non-trivial constraints since negated user-defined constraints are ignored.

4 Solver Optimization

To improve the runtime efficiency of the solvers and readability of the rules, redundant guard entailment checks are removed. This is achieved by expanding the guard expressions to disjunctive normal form and splitting each disjunct into a new rule. Then, we apply the redundant rules removal algorithm of [9] on the complete rule set. After redundant rules are removed, guards originating from the same rule are recombined to avoid loss of completeness.

4.1 Guard Splitting

The symbolic construction approach constructs rules of the form:

$$p(\bar{x}) \Leftrightarrow \bigwedge_{j=1,j\neq i}^{n} \bigvee_{k=1}^{|x|+m_j} E_j^k \mid \bar{x}{=}\bar{t}_i \wedge C_i$$

where

$$E_j^k{=}\forall \bar{y}_j^k \left(x^k{\neq}t_j^k\right) \qquad \text{for } k \in \{1,\ldots,|x|\},$$

$$E_j^{|x|+k}{=}\exists \bar{y}_j \left(\bar{x}{=}\bar{t}_j \wedge \neg c_j^k\right) \qquad \text{for } k \in \{1,\ldots,m_j\},$$

the $\bar{y}_j^k$ is the sequence of variables in the term t_j^k and m_j is the number of built-in constraints in C_j.

To split the guard into rules (Fig. 2), we distribute the conjunction over the disjunction and get a formula in disjunctive normal form where the number of disjuncts is $\Pi_{j=1,j\neq i}^{n}|x| + m_j$. Then each disjunct is simplified to an equivalent conjunction of constraints by the available built-in solver and superfluous disjuncts removed. These include multiple occurrences of a disjunct (irrespective of the order of constraints within the disjunct) and *false*. Each simplified disjunct is split into a new rule.

Example 5 (Guard Splitting). Consider the previously constructed rule of lex:

$$\begin{aligned} lex(L_1, L_2) \Leftrightarrow\ & \\ & (L_1{=}[] \ \vee\ L_2{=}[] \ \vee\ (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2)) \wedge \\ & (L_1{=}[] \ \vee\ L_2{=}[] \ \vee\ (L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4)) \\ & \mid L_1{=}[] \wedge L_2{=}[] \end{aligned}$$

begin
$Rules_{in}$: initial rule set
$Rules_{out}$={}: resultant rule set

while $Rules_{in}$≠{} **do**
Remove from $Rules_{in}$ an element denoted R
R is of the form $p \Leftrightarrow E \mid D$
G is the cartesian product of the $n-1$ conjuncts of the guard E

while G≠{} **do**
Remove from G an element denoted G_e
G_{simp}: the result of executing G_e by the built-in solver
if G_{simp}≠*false* **then**
$Rules_{out}$=$Rules_{out} \cup \{p \Leftrightarrow G_{simp} \mid D\}$
end if
end while
end while
end

Fig. 2. Guard Splitting

Transforming the guard expression to disjunctive normal form, we get:

$(L_1{=}[] \wedge L_1{=}[]) \vee$
$(L_1{=}[] \wedge L_2{=}[]) \vee$
$(L_1{=}[] \wedge L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4) \vee$
$(L_2{=}[] \wedge L_1{=}[]) \vee$
$(L_2{=}[] \wedge L_2{=}[]) \vee$
$(L_2{=}[] \wedge L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4) \vee$
$(L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2 \wedge L_1{=}[]) \vee$
$(L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2 \wedge L_2{=}[]) \vee$
$(L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2 \wedge L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4)$

To simplify the resultant expression, each disjunct is executed by the built-in constraints solver and superfluous disjuncts removed. We assume that for the conjunction of constraints, the built-in solver:

- Removes identical occurrences of constraints
- Simplifies constraints (e.g. $L{=}[] \wedge L{=}[X|T] \Leftrightarrow$ *false* and $X{\geq}Y \wedge X{\neq}Y \Leftrightarrow X{>}Y$)
- Propagates new constraints (e.g. $L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \Rightarrow X_1{=}X_3 \wedge T_1{=}T_3 \wedge X_2{=}X_4 \wedge T_2{=}T_4$)

The expression simplifies to:

$(L_1{=}[]) \vee (L_1{=}[] \wedge L_2{=}[]) \vee (L_2{=}[]) \vee (L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{>}X_2)$

which splits into the following rules:

$$lex(L_1, L_2) \Leftrightarrow L_1{=}[] \mid L_1{=}[] \wedge L_2{=}[]$$
$$lex(L_1, L_2) \Leftrightarrow L_1{=}[] \wedge L_2{=}[] \mid L_1{=}[] \wedge L_2{=}[]$$
$$lex(L_1, L_2) \Leftrightarrow L_2{=}[] \mid L_1{=}[] \wedge L_2{=}[]$$
$$lex(L_1, L_2) \Leftrightarrow L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{>}X_2 \mid L_1{=}[] \wedge L_2{=}[]$$

4.2 Redundant Rules Removal

To remove redundant rules, the algorithm of [9] is used. The idea of the algorithm is based on operational equivalence of programs. The algorithm (Fig. 3) basically checks if the computation step due to a rule can be performed by the remainder of the program. It determines this by executing the head and guard in both the program and the program without the rule in it. If the results are identical upto renaming of variables and logical equivalence of built-in constraints), then the rule is obviously redundant and can be removed.

---

begin
  $Rules_{in}$: the initial rule set
  $Rules_{out}$: the resultant rule set without redundancy
  $Rules_{out}{=}Rules_{in}$

  while $Rules_{in}{\neq}\{\}$ **do**
    Remove from $Rules_{in}$ an element denoted R
    lhs: the head and guard of the rule R
    S_1: the result of executing lhs in $Rules_{out}$
    $Rules_{remaining}{=}Rules_{out}\backslash\{R\}$
    S_2: the result of executing lhs in $Rules_{remaining}$
    if S_1 is identical to S_2 **then**
      $Rules_{out}{=}Rules_{remaining}$
    end if
  end while
end

Fig. 3. Redundancy Removal Algorithm

Example 6 (Redundant Rules Removal). Consider the following two rules of lex:

$$lex(L_1, L_2) \Leftrightarrow L_1{=}[] \mid L_1{=}[] \wedge L_2{=}[]$$
$$lex(L_1, L_2) \Leftrightarrow L_1{=}[] \wedge L_2{=}[] \mid L_1{=}[] \wedge L_2{=}[]$$

The second rule is redundant since its operation is covered by the first rule. Removing the second rule from the rule set and querying the remaining set with

its head and guard $\langle A{=}[] \wedge B{=}[] \wedge lex(A,B)\rangle$, the first rule is applied and gives the same result $\langle A{=}[] \wedge B{=}[]\rangle$ as the second rule.

After redundant rules are removed, guards originating from the same rule are recombined to avoid loss of completeness. Further guard optimization techniques have been addressed in [10].

5 Combined Approach

To improve the filtering capabilities of our constructed solvers, we propose extending our solvers with rules generated by the orthogonal approach "generate and test" of [1]. To reduce the search space of the generate and test method, the symbolic construction algorithm is run first and the constructed rules eliminated from the enumeration tree of the generate and test. Moreover, the algorithm of [9] is used to remove the redundant rules of the combined solver. In general, the combined solvers are more expressive than the solvers of either approaches.

Example 7 (Combined Approach). The combined solver for *lex* is given below. The first three rules represent the solver obtained from the symbolic construction approach and the last rule is added by the generate and test.

$$lex(L_1,L_2) \Leftrightarrow L_1{=}[] \vee L_2{=}[] \mid L_1{=}[] \wedge L_2{=}[] \quad (1)$$

$$lex(L_1,L_2) \Leftrightarrow L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{\neq}X_4 \mid L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{<}X_2 \quad (2)$$

$$lex(L_1,L_2) \Leftrightarrow L_1{=}[X_1|T_1] \wedge L_2{=}[X_2|T_2] \wedge X_1{\geq}X_2 \mid L_1{=}[X_3|T_3] \wedge L_2{=}[X_4|T_4] \wedge X_3{=}X_4 \wedge lex(T_3,T_4) \quad (3)$$

$$lex([X_1|T_1],[X_2|T_2]) \Rightarrow X_1{\leq}X_2 \quad (4)$$

The solver is sound. All rules are logical consequences of the constraint definition.

The solver terminates. The interesting case for termination is the recursive rule (3). The ranking function for $lex(L_1,L_2)$ is defined as the positive combination of the rank of its arguments:

$$rank(lex(L_1,L_2)) = length(L_1) + length(L_2)$$

The length of a list is expressed in the ranking function scheme as:

$$length([]) = 0$$
$$length([H|T]) = 1 + length(T)$$

All other constraints in the rule are built-ins and are ranked as 0. The rule terminates since the rank of the head and guard is greater than that of its body:

$$rank(lex([X_1|T_1],[X_2|T_2])) > rank(lex(T_1,T_2))$$

The solver is not propagation complete. In [4] the below complete *lex* solver was presented:

$$lex([],[]) \Leftrightarrow true \tag{5}$$
$$lex([X_1|T_1],[X_2|T_2]) \Leftrightarrow X_1{<}X_2 \mid true \tag{6}$$
$$lex([X_1|T_1],[X_2|T_2]) \Leftrightarrow X_1{=}X_2 \mid lex(T_1,T_2) \tag{7}$$
$$lex([X_1|T_1],[X_2|T_2]) \Rightarrow X_1{\leq}X_2 \tag{8}$$
$$lex([X_1,U|T_1],[X_2,V|T_2]) \Leftrightarrow U{>}V \mid X_1{<}X_2 \tag{9}$$
$$lex([X_1,U|T_1],[X_2,V|T_2]) \Leftrightarrow U{\geq}V \wedge T_1{=}[\_|\_] \mid lex([X_1,U],[X_2,V]) \wedge lex([X_1|T_1],[X_2|T_2]) \tag{10}$$

The solver consists of three pairs of rules: the first two correspond to base cases of the recursion, the middle two perform forward reasoning, and the last two perform backward reasoning. By comparison we find that the backward reasoning rules are not subsumed by our combined *lex* solver rendering the solver incomplete.

Consider the query $\langle lex([A_1,A_2,A_3,A_4],[B_1,B_2,B_3,B_4])\rangle$ where the domains of the variables are defined as follows:

$$A_1{=}\{1,3,4\}, A_2{=}\{1,2,3,4,5\}, A_3{=}\{1,2\}, A_4{=}\{3,4,5\}$$
$$B_1{=}\{1\}, B_2{=}\{0,1,2,3,4\}, B_3{=}\{0,1\}, B_4{=}\{0,1,2\}$$

In the case of the combined solver for *lex*, rule (3) is fired since $A_1{\geq}B_1$ enforcing equality on the values of A_1 and B_1 before calling *lex* recursively on the remaining list elements. The relation between A_2 and B_2 satisfies none of the guards, thus rule (4) is fired which enforces $A_2{\leq}B_2$. The solution becomes:

$$A_1{=}\{1\}, A_2{=}\{1,2,3,4\}, A_3{=}\{1,2\}, A_4{=}\{3,4,5\},$$
$$B_1{=}\{1\}, B_2{=}\{1,2,3,4\}, B_3{=}\{0,1\}, B_4{=}\{0,1,2\},$$
$$A_2{\leq}B_2, lex([A_2,A_3,A_4],[B_2,B_3,B_4])$$

In the case of the *lex* solver of [4], rules (8), (7), (10), and (9) are applied in that order and further constrain the domains of the variables to:

$$A_1{=}\{1\}, A_2{=}\{1,2,3\}, A_3{=}\{1,2\}, A_4{=}\{3,4,5\},$$
$$B_1{=}\{1\}, B_2{=}\{2,3,4\}, B_3{=}\{0,1\}, B_4{=}\{0,1,2\},$$
$$A_2{<}B_2, lex([A_2,A_3],[B_2,B_3])$$

6 Conclusion

In this paper we have presented an algorithm that automatically constructs rule-based solvers from the constraint definition. The algorithm is an orthogonal approach to the general direction of the work done in the field as it is based on symbolic construction rather than a generate and test method. Contrary to

other approaches, given a recursive constraint definition the algorithm is able to generate recursive rules which allow reasoning over arguments of arbitrary length.

The constructed solvers are a good basis for constraint reasoning and can be extended manually or with rules generated using other approaches. We have proposed extending our rules with those generated by the algorithm in [1]. In general, the solvers generated using the combined approach are more expressive than those generated by either of the two approaches acting alone.

An interesting direction for future work to improve the expressiveness of the generated solvers is to incorporate negated user-defined constraints in the symbolic construction approach.

Acknowledgments. We would like to thank Thom Frühwirth, Frank Raiser, Jon Sneyers and the anonymous reviewers for valuable comments on a preliminary version of this paper.

References

1. Abdennadher, S., Rigotti, C.: Automatic Generation of CHR Constraint Solvers. Journal of Theory and Practice of Logic Programming (TPLP) **5**(4-5) (2005) 403–418
2. Carlsson, M., Beldiceanu, N.: Revisiting the Lexicographic Ordering Constraint. In: Technical Report T2002-17. Swedish Institute of Computer Science (2002)
3. Frisch, A., Hnich, B., Kzltan, Z., Miguel, I., Walsh, T.: Global Constraints for Lexicographic Orderings. In: International Conference on Principles and Practice of Constraint Programming, CP2002. Volume 2470 of Lecture Notes in Computer Science., Springer (2002) 93–108
4. Frühwirth, T.: Complete Propagation Rules for Lexicograhic Order Constraints over Arbitary Domains. In: Recent Advances in CHR. Volume 3978 of Lecture Notes in Computer Science., Springer (2006) 14–28
5. Abdennadher, S., Sobhi, I.: Generation of Rule-based Constraint Solvers: Combined Approach. In: International Symposium in Logic-Based Program Synthesis and Transformation, LOPSTR07, Revised Selected Papers. Volume 4915 of Lecture Notes in Computer Science., Springer (2008) 106–120
6. Clark, K.: Negation as Failure. In: Logic and Databases. Plenum Press, New York (1978) 293–322
7. Frühwirth, T.: Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. Journal of Logic Programming **37**(1-3) (1998) 95–138
8. Frühwirth, T.: Proving Termination of Constraint Solver Programs. In: New Trends in Constraints. Volume 1865 of Lecture Notes in Computer Science., Springer (2000) 298–317
9. Abdennadher, S., Frühwirth, T.: Integration and Optimization of Rule-based Constraint Solvers. In: International Symposium on Logic-based Program Synthesis and Transformation, LOPSTR03. Volume 3018 of Lecture Notes in Computer Science., Springer (2004) 198–213
10. Sneyers, J., Schrijvers, T., Demoen, B.: Guard and Continuation Optimization for Occurrence Representations of CHR. In: International Conference on Logic Programming, ICLP05. Volume 3668 of Lecture Notes in Computer Science., Springer (2005) 83–97