Complete Propagation Rules for Lexicographic Order Constraints over Arbitrary Domains

Thom Frühwirth

Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. We give an efficiently executable specification of the global constraint of lexicographic order in the Constraint Handling Rules (CHR) language. In contrast to previous approaches, the implementation is short and concise without giving up on the best known worst case time complexity. It is incremental and concurrent by nature of CHR. It is provably correct and confluent. It is independent of the underlying constraint system, and therefore not restricted to finite domains. We have found a direct recursive decomposition of the problem. We also show completeness of constraint propagation, i.e. that all possible logical consequences of the constraint are generated by the implementation. Finally, we report about some practical implementation experiments.

1 Introduction

Lexicographic orderings are common in everyday life as the alphabetical order used in dictionaries and listings (e.g., 'zappa' comes before 'zilch'). In computer science, lexicographic orders also play a central role in termination analysis, for example for rewrite systems [3]. In constraint programming, these orders have recently raised interest because of their use in symmetry breaking (e.g. [14]) and earlier in modelling preferences among solutions (e.g. [7]).

A natural question to ask is whether lexicographic orders can be implemented as constraints and what would be appropriate propagation algorithms. There are two approaches to this problem, starting with [8] and [6]. Both consider the case of finite domain constraints and (hyper/generalized) arc consistency algorithms, while our work is independent of the underlying constraint system and achieves complete constraint progagation as well. All approaches, including ours, yield algorithms with a worst-case time complexity that is linear in the size of the lexicographic ordering constraint.

The algorithms and their derivation are quite different, however. In [8] an algorithm based on two pointers that move along the elements of the sequences to be lexicographically ordered is given. The algorithm's description consists of five procedures with 45 lines of pseudo-code. In [6], a case analysis of the lexicographic order constraints yields 7 cases to distinguish, these are translated into a finite automaton that is then made incremental. The pseudo-code of the algorithm has 42 lines [5]. The manual derivation of the algorithm is made semi-automatic in a subsequent paper [4], that can deal with an impressive range

of global constraints over sequences. The pseudo-code of a simple constraint checker is converted by hand into a corresponding automaton code (16 lines) that is automatically translated into automata constraints that allow incremental execution of the automaton and so enforce arc consistency. Note that the automaton code is interpreted at run-time.

We summarize that these approaches are based on imperative pseudo-code that seems either lengthy or requires subsequent translation into a different formalism. Their specifications seem hard to analyse and are not directly executable. In contrast, we give a short and concise executable specification in the Constraint Handling Rules (CHR) language that consists of 6 rules that derive from three cases. The problem is solved by recursive decomposition, no additional constraints need to be defined. The implementation is incremental and concurrent by nature of CHR. It is independent of the underlying constraint system, and therefore not restricted to finite domains. Its CHR rules can be analysed, for example we will show their confluence using a confluence checker, and prove their logical correctness. We derive worst-case time complexity that is parameterized by the cost of handling built-in constraints. We also show that the rules are complete, that they propagate as much information (constraints) as possible.

CHR [9, 13, 16] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) into simpler ones until they are solved. CHR was initially developed for writing constraint solvers, but has matured into a general-purpose concurrent constraint language over the last decade. Its main features are a kind of multi-set rewriting combined with propagation rules. The clean logical semantics of CHR facilitates non-trivial program analysis and transformation. Implementations of CHR now exist in many Prolog systems, also in Haskell and Java. Besides constraint solvers, applications of CHR range from type systems and time tabling to ray tracing and cancer diagnosis.

Overview of the Paper. After introducing CHR, we give our generic implementation of the global constraint for lexicographic orderings in Section 3. Then, in separate sections, we discuss confluence, logical correctness, completeness, worst-case time complexity and some implementation experiments before we conclude. This paper is a significantly revised and extended version of [11].

2 Preliminaries: Constraint Handling Rules

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [9, 13, 16]. Readers familiar with CHR can skip this section.

2.1 Syntax of CHR

We distinguish between two different kinds of constraints: *built-in (pre-defined) constraints* which are solved by a given constraint solver, and *CHR (user-defined)*

constraints which are defined by the rules in a CHR program. This distinction allows one to embed and utilize existing constraint solvers as well as side-effect-free host language statements. Built-in constraint solvers are considered as black-box in whose behavior is trusted and that do not need to be modified or inspected. The solvers for the built-in constraints can be written in CHR itself, giving rise to a hierarchy of solvers [15].

Definition 1. A *CHR program* is a finite set of rules. There are two main kinds of rules:

$$\begin{array}{ll} \textit{Simplification rule:} & \textit{Name} \ @ \ H \Leftrightarrow C \mid B \\ \textit{Propagation rule:} & \textit{Name} \ @ \ H \Rightarrow C \mid B \end{array}$$

Name is an optional, unique identifier of a rule, the *head* H is a non-empty conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a conjunction of built-in and CHR constraints. A trivial guard expression "*true* |" can be omitted from a rule.

Example 1. For example, let $\leq$ be a built-in constraint symbols with the usual meaning. Here is a rule for a CHR constraint `max`, where `max(X,Y,Z)` means that `Z` is the maximum of `X` and `Y`:

```
max(X,Y,Z) ⇔ X≤Y | Z =Y.
```

2.2 Declarative Semantics of CHR

The CHR rules have an immediate logical reading, where the guard implies a logical equality or implication between the l.h.s. and r.h.s. of a rule.

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

$$\forall(C \rightarrow (H \leftrightarrow \exists \bar{y}\ B)),$$

where $\forall$ denotes universal closure as usual and $\bar{y}$ are the variables that appear only in the body B.

The logical meaning of a propagation rule is an implication provided the guard holds

$$\forall(C \rightarrow (H \rightarrow \exists \bar{y}\ B)).$$

The logical meaning $\mathcal{P}$ of a CHR program P is the conjunction of the logical meanings of its rules united with a consistent *constraint theory* CT that defines the built-in constraint symbols.

Example 2. Recall the rule for `max` from Example 1. The rule means that `max(X,Y,Z)` is logically equivalent to `Z=Y` if `X≤Y`:

$$\forall(X \leq Y \rightarrow (max(X,Y,Z) \leftrightarrow Z{=}Y))$$

2.3 Operational Semantics of CHR

At runtime, a CHR program is provided with an initial state and will be executed until either no more rules are applicable or a contradiction occurs.

The operational semantics of CHR is given by a transition system (Fig. 1). Let P be a CHR program. We define the transition relation $\mapsto$ by two computation steps (transitions), one for each kind of CHR rule. *States* are goals, i.e. conjunctions of built-in and CHR constraints. States are also called *(constraint) stores*. In the figure, all upper case letters are meta-variables that stand for conjunctions of constraints. The constraint theory CT defines the semantics of the built-in constraints. G_{bi} denotes the built-in constraints of G.

Simplify

If	$(r@H \Leftrightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule named r in P
and	$CT \models \forall\, (G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_r (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If	$(r@H \Rightarrow C \mid B)$ is a fresh variant with variables $\bar{x}$ of a rule named r in P
and	$CT \models \forall\, (G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto_r (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Fig. 1. Computation Steps of Constraint Handling Rules

Starting from an arbitrary *initial goal (state, query, problem)*, CHR rules are applied exhaustively, until a fixpoint is reached. A *final state (answer, solution)* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent.

A simplification rule $H \Leftrightarrow C \mid B$ *replaces* instances of the CHR constraints H by B provided the guard C holds. A propagation rule $H \Rightarrow C \mid B$ instead *adds* B to H. If new constraints arrive, rules are reconsidered for application. Computation stops if the built-in constraints become inconsistent. Trivial non-termination of the **Propagate** computation step is avoided by applying a propagation rule at most once to the same constraints (see the more concrete semantics in [1]).

In more detail, a rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is logically implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice.

Example 3. Here are some sample computations involving the rule for `max`:

$\texttt{max}(1,2,\texttt{M}) \mapsto \texttt{M}{=}2.$
$\texttt{max(A,B,M)} \wedge \texttt{A}{<}\texttt{B} \mapsto \texttt{M}{=}\texttt{B} \wedge \texttt{A}{<}\texttt{B}.$
$\texttt{max}(\texttt{A},\texttt{A},\texttt{M}) \mapsto \texttt{M}{=}\texttt{A}.$

3 The Lexicographic Order Constraint Solver

A lexicographic order allows one to compare sequences by pairwise comparing the elements of the sequences.

Definition 3. Given two sequences l_1 and l_2 of variables of the same length n, $[x_1, \ldots, x_n]$ and $[y_1, \ldots, y_n]$. Then l_1 is lexicographically smaller than or equal to l_2, written $l_1 \preceq_{lex} l_2$, iff either $n{=}0$ or $x_1{<}y_1$ or $x_1{=}y_1$ and $[x_2, \ldots, x_n] \preceq_{lex} [y_2, \ldots, y_n]$.

The corresponding logical specification of the `lex` constraint thus is:

$$\begin{aligned} l_1 \preceq_{lex} l_2 \leftrightarrow\ & (l_1{=}[] \wedge l_2{=}[]) \vee \\ & (l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{<}y) \vee \\ & (l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{=}y \wedge l_1' \preceq_{lex} l_2') \end{aligned}$$

In our CHR solver for the `lex` constraint we will use concrete syntax of Prolog implementations of CHR. Variables start with upper-case letters, constraint and function symbols with lower-case letters. Lists are enclosed in square brackets, with their elements seperated by commata, while after the symbol `''|''` the remainder of the list follows as a list. Conjunction $\wedge$ is written as comma `','`.

Our solver will be independent from the constraint system in which the built-in constraints (inequalities) are defined. Different list elements can be from different constraint domains if their inequalities are polymorphic. They can even be a (differently named) lexicographic constraint provided it is built-in.

The derivation of the following six rules for our lexicographic order constraint solver is explained in [11]. The solver consists of three pairs of rules, the first two corresponding to base cases of the recursion (garbage collection), then two rules performing forward reasoning (recursive traversal and implied inequality), and finally two for backward reasoning, covering a not so obvious special case when the lexicographic constraint has a unique solution.

```
l1 @ [] lex [] <=> true.
l2 @ [X|L1] lex [Y|L2] <=> X<Y | true.
l3 @ [X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
l4 @ [X|L1] lex [Y|L2] ==> X=<Y.

l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
l6 @ [X,U|L1] lex [Y,V|L2] <=> U>=V, L1=[_|_] |
                               [X,U] lex [Y,V], [X|L1] lex [Y|L2].
```

The first three rules `l1`, `l2` and `l3` are directly derived from the three disjuncts of the logical specification. The notation `[X|L]` refers to a list with first element `X` and the remainder of the list is the list `L`. The three rules will apply when the lists are empty or when the relationship between the leading list elements `X` and `Y` is sufficiently known. The built-in constraints `X<Y` and `X=Y` are in the guards, so they check if the appropriate relationship between the variables holds. When a rule is tried, the built-in constraint solver has to check if the guard is implied by the current built-in constraints.

For example, the three queries `[1] lex [2]`, `[X] lex [X]`, and `[X] lex [Y], X<Y` will all reduce to `true`. For finite domains, consider `X in {0,1}, Y in {2,3}, [X] lex [Y]`. Rule `l2` asks in the guard if the constraint `X<Y` holds, i.e. if it is implied by the current built-in constraints. If the built-in finite domain solver is strong enough to infer `X<Y` from `X in {0,1}, Y in {2,3}`, then rule `l2` is applicable and its application results in `X in {0,1}, Y in {2,3}`. For simplicity, we will just use explicit inequalities in our examples.

The propagation rule `l4` implements a common consequence of the last two disjuncts of the logical specification. The built-in inequality constraint appears in the body of the rule and is thus enforced when the rule is applied.

For example, to the query `[R|Rs] lex [T|Ts], R<>T` only the propagation rule is applicable and adds `R=<T`. This results in `[R|Rs] lex [T|Ts], R<T` after simplification of the built-in constraints for inequality. Now rule `l2` is applicable, the `lex`-constraint is removed and the final answer is the remaining `R<T`.

Rule `l5` deals with the special case where the elements of the second pair of the sequence are related by a strict inequality in the wrong way such that the only (way to a) solution is to enforce a strict inequality on the first two elements. Note that rules `l4` and `l5` are the only ones that directly impose a built-in constraint. Rule `l6` uses double recursion, but note that the first recursive `lex` constraint has a fixed, small list length. The rule deals with the case where the wrong inequality treated in `l5` is further down the lists. The additional condition `L1=[_|_]` in the guard of rule `l6` avoids non-termination in case `L1=[]`.

To see how rules `l5` and `l6` work together, consider the query `[R1,R2,R3] lex [T1,T2,T3], R2>=T2, R3>T3`. Since `R2>=T2`, rule `l6` is applicable, and leads to `R2>=T2, R3>T3, [R1,R2] lex [T1,T2], [R1,R3] lex [T1,T3]`. Now rule `l5` can be applied to the second `lex` constraint, and we arrive at `R2>=T2, R3>T3, [R1,R2] lex [T1,T2], R1<T1`. Because now `R1<T1` is enforced, rule `l2` removes the remaining `lex` constraint and the final answer is `R2>=T2, R3>T3, R1<T1`.

4 Confluence

Typically, CHR programs for constraint solving are well-behaved, i.e. terminating and confluent. Confluence means that the result of a computation is independent from the order in which rules are applied to the constraints. This also implies that the order of constraints in a goal does not matter. Once termination has been established [10], there is a decidable, sufficient and necessary test for confluence [1, 2]. In the latter papers it is also shown that confluent CHR programs have a consistent logical reading.

Definition 4. A CHR program is *confluent* if for all computation states S, S_1, S_2: If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

For checking confluence, one takes copies (with fresh variables) of two rules (not necessarily different) from the program. The heads of the rules are *overlapped* by equating at least one head constraint from one rule with one from the other rule. For each overlap, one considers the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If a critical pair is not joinable, one has found a counterexample for confluence of the program.

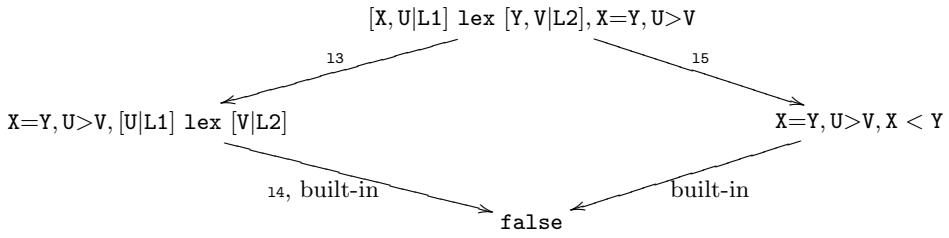
We used and improved the confluence checker mentioned in [12] to check confluence of the `lex` constraint. The six rules for the lexicographic order constraint are confluent, the program code and its results are available at:
`www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/conflexico.pl`

The rule `l1` cannot give rise to any critical pair. It does not overlap with any other rule, since it is the only one dealing with empty lists. The rules `l2` and `l3` are mutually exclusive. There are overlaps between all the remaining pairs of rules. If rule `l2` or rule `l4` is dropped, the solver becomes non-confluent, while the other rules can be dropped without hurting confluence.

Example 4. Consider the overlap between the rules

```
l3 @ [X|L1] lex [Y|L2] <=> X=Y | L1 lex L2.
l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
```

which is `[X,U|L1] lex [Y,V|L2], U>V, X=Y` and which leads to the following confluence check:



Using the first rule, we arrive at `X=Y, U>V, [U|L1] lex [V|L2]`. Using the second rule, we arrive at `X=Y, U>V, X<Y`. These two states form the critical pair. The propagation rule `l4` is applicable to the first state `X=Y, U>V, [U|L1] lex [V|L2]` and leads to `X=Y, U>V, [U|L1] lex [V|L2], U=<V`, which fails due to the contradicting constraints on `U` and `V`. The second state immediately fails due to the contradicting constraints on `X` and `Y`. Hence, this critical pair is joinable, in both cases we finally fail (independent of the order of rule applications).

Example 5. Another confluence check involves the rules `l5` and `l6`.

```
l5 @ [X,U|L1] lex [Y,V|L2] <=> U>V | X<Y.
l6 @ [X,U|L1] lex [Y,V|L2] <=> U>=V, L1=[_|_] |
                               [X,U] lex [Y,V], [X|L1] lex [Y|L2].
```

Their overlap is

```
[X,U|L1] lex [Y,V|L2], U>V, L1=[_|_].
```

The resulting critical pair is

```
U>V, L1=[_|_], X<Y   vs.
U>V, L1=[_|_], [X,U] lex [Y,V], [X|L1] lex [Y|L2].
```

The first state of the critical pair is already a final state, in the second one, rule `l5` can be applied to the first `lex` constraint resulting in `U>V, L1=[_|_], X<Y, [X|L1] lex [Y|L2]`. Now, since `X<Y`, rule `l2` can be applied to remove the remaining `lex` constraint, the two states of the critical pair are joinable.

5 Logical Correctness

CHR programs can be formally verified on the basis of their logical reading. Recall that the logical meaning of a CHR program is the logical meaning of its rules united with the constraint theory CT for the built-in constraints.

Definition 5. Let $\mathcal{P}$ be the logical meaning of a CHR program P. Let $\mathcal{S}$ be a *logical specification* for P, i.e. a consistent theory for the CHR constraints in P. Then program P is *logically correct* with respect to specification $\mathcal{S}$ iff

$$\mathcal{S} \cup CT \models \mathcal{P}.$$

The logical reading of the six rules for the lexicographic order constraint solver is as follows.

$$\begin{array}{rl}
 & ([] \preceq_{lex} []) \\
X{<}Y \rightarrow & ([X|L1] \preceq_{lex} [Y|L2]) \\
X{=}Y \rightarrow & ([X|L1] \preceq_{lex} [Y|L2] \quad \leftrightarrow L1 \preceq_{lex} L2) \\
 & ([X|L1] \preceq_{lex} [Y|L2] \quad \rightarrow X \leq Y) \\
U{>}V \rightarrow & ([X,U|L1] \preceq_{lex} [Y,V|L2] \leftrightarrow X{<}Y) \\
(U \geq V \wedge L1{=}[_|_]) \rightarrow & ([X,U|L1] \preceq_{lex} [Y,V|L2] \leftrightarrow \\
 & \quad ([X,U] \preceq_{lex} [Y,V] \wedge [X|L1] \preceq_{lex} [Y|L2]))
\end{array}$$

For logical correctness, we have to show that these formulas are logical consequences of the logical specification given by

$$\begin{array}{rl}
l_1 \preceq_{lex} l_2 \leftrightarrow & (l_1{=}[] \wedge l_2{=}[]) \vee \\
 & (l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{<}y) \vee \\
 & (l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{=}y \wedge l_1' \preceq_{lex} l_2')
\end{array}$$

For example, it is easy to see that the logical reading of propagation rule `14` is a common consequence of the last two disjuncts of the specification,

$$(l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{<}y) \ \vee \ (l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x{=}y \wedge l_1' \preceq_{lex} l_2') \quad \rightarrow$$

$$l_1{=}[x|l_1'] \wedge l_2{=}[y|l_2'] \wedge x \leq y.$$

Proof for Rule `l6`. As a more involved example, we prove that the logical reading of propagation rule `l6` is a logical consequence of the specification. From the specification it follows

$$[X|L1]\preceq_{lex}[Y|L2] \leftrightarrow (X{<}Y \ \vee \ X{=}Y \wedge L1\preceq_{lex}L2)$$

For the rule `l6`, we will actually prove a slightly stronger result by removing $L1{=}[\_|\_]$ from the precondition (the condition was introduced to ensure termination). Instead of $C \rightarrow (H \leftrightarrow B)$ we use the logically equivalent $(H \wedge C) \leftrightarrow (C \wedge B)$).

To show the equivalence of the l.h.s. and r.h.s. of the formula, we will now replace the `lex` constraints in the logical reading of the rule according to the specification, distribute conjunction over disjunction and simplify by removing unsatisfiable disjuncts.

The l.h.s. of the rule, $[X,U|L1]\preceq_{lex}[Y,V|L2] \wedge U{\geq}V$, becomes

$$U{\geq}V \wedge (X{<}Y \ \vee \ X{=}Y \wedge U{<}V \ \vee \ X{=}Y \wedge U{=}V \wedge L1\preceq_{lex}L2) \leftrightarrow$$
$$(U{\geq}V \wedge X{<}Y \ \vee \ X{=}Y \wedge U{=}V \wedge L1\preceq_{lex}L2)$$

The r.h.s. $U{\geq}V \wedge [X,U]\preceq_{lex}[Y,V] \wedge [X|L1]\preceq_{lex}[Y|L2]$ becomes

$$U{\geq}V\wedge(X{<}Y \vee X{=}Y\wedge U{<}V \vee X{=}Y\wedge U{=}V)\wedge(X{<}Y \vee X{=}Y\wedge L1\preceq_{lex}L2) \leftrightarrow$$
$$U{\geq}V\wedge(U{\geq}V\wedge X{<}Y \vee X{=}Y\wedge U{=}V)\wedge(U{\geq}V\wedge X{<}Y \vee U{\geq}V\wedge X{=}Y\wedge L1\preceq_{lex}L2)$$
$$\leftrightarrow (U{\geq}V \wedge X{<}Y \ \vee \ X{=}Y \wedge U{=}V \wedge L1\preceq_{lex}L2)$$

Both sides, l.h.s. and r.h.s., are equivalent.

6 Worst-Case Time Complexity

We would like to give a complexity result that is independent from the constraint system in which the built-in constraints (inequalities) are defined. The reason is that most constraint systems, such as Booleans, finite domains, and linear polynomials, admit these inequalities, but the typical algorithms used (e.g. arc and path consistency, simplex) have different time complexities and achieve different degrees of completeness (local or global). We therefore give our complexity result in the number of atomic built-in constraints that are checked and imposed, respectively.

Lemma 1. For the rules of the `lex` constraint, the number of checks and additions of built-in constraints is proportional to the number of rule applications.

Proof. Head matching can be done in constant time, guards contain at most one built-in inequality constraint to check, and rule bodies directly impose at most one built-in inequality constraint.

We show now that an upper bound on the number of rule applications r depends on the list lengths only. We treat `lex` constraints with list arguments up to two elements separately, because they play a special role in rule `l6`.

Lemma 2. The number of checks and additions of built-in inequality constraints is linear in the length of the list.

Proof. By Lemma 1, it suffices to consider the number of rule applications. We use the following recurrence equations generated from the rules of the `lex` constraint solver. The number of rule applications involving a list of some given length is computed as follows in the equations below: We charge 1 (unit) cost for applying one of the applicable rules and add the number of rule applications caused by the body of the respective rule. The unit costs are represented by constants $l1$ to $l6$ to indicate which rule is applied. From all the potential rule applications, the maximum is taken. Since the propagation rule is always applicable to non-empty lists, its cost is added outside of the maximum expression.

$$\begin{array}{l|l} & l1 = l2 = l3 = l4 = l5 = l6 = 1 \\ \hline r(0) & = l1 = 1 \\ r(1) & = max(l2, l3{+}r(0), l5){+}l4 = 3 \\ r(2) & = max(l2, l3{+}r(1), l5){+}l4 = 5 \\ & \vdots \\ r(n) & = max(l2, l3{+}r(n{-}1), l5, l6{+}r(2){+}r(n{-}1)){+}l4 = 7{+}r(n{-}1) < 7n{-}8 \end{array}$$

To empty lists, only the rule `l1` is applicable. To lists with one or two elements, the rules `l2` up to `l5` are applicable, but not rule `l6`. The propagation rule `l4` can be applied at most once to each `lex` constraint. The recursive rules `l3` and `l6` dominate the costs.

For lists of length n less than or equal to 2, the number of rule applications is bounded by a constant (at most 5). For lists of length greater than 2, the number of rule applications is linear in the length of the list.

For the overall complexity, we should not forget about waking: If a variable of a pending `lex` constraint gets more constrained by a built-in constraint, the `lex` constraint will be woken. Then the results hold even if the built-in constraints are imposed incrementally, as is standard in constraint programming applications.

Theorem 1. The overall worst case time complexity is linear modulo the cost of handling the built-in constraints. At most $O(n + w)$ built-in constraints are checked, imposed or woken where n is the list length and w is the number of wake (propagation) events caused by the built-in constraint solver.

Proof. The result follows from Lemma 2 and the following observation: If a CHR constraint is woken, it's rules will be re-checked for applicability. If a rule is applicable, the cost of the continuation of the computation on the `lex` constraint has already been accounted for in the above calculations. But what is the cost of waking `lex` in vain, i.e. if no rule turns out to be applicable? Then a constant number of head matchings and guard checks has been performed if rules are tried.

7 Completeness

In this section we discuss completeness of the constraint solver for the lexicographic order constraint, i.e. if it produces all built-in constraints, i.e. inequalities, that logically follows from the `lex` constraint and some given inequalities.

We already know that the solver is correct and confluent. Thus it cannot propagate incorrect conctraints and starting from a given goal it will always propagate the same constraints, no matter which of the applicable rules are applied. Thus what is left to show for completeness is that all possible propagations are performed, not just a few.

Of course, also the completeness result is relative to the built-in constraint solver. In particular, if its entailment check is too weak to detect all cases where guard inequalities are implied, the `lex` constraint solver will also become incomplete. This is the case for finite domains, since the underlying arc consistency algorithm only provides local completeness.

Definition 6. A *solution* of a lexicographic order constraint $[x_1, \ldots, x_n] \preceq_{lex} [y_1, \ldots, y_n]$ is of the form

$$x_1{=}y_1 \wedge x_2{=}y_2 \wedge \ldots \wedge x_{i-1}{=}y_{i-1}[\wedge x_i{<}y_i] \quad (1 \leq i \leq n{+}1),$$

where $x_i{<}y_i$ is dropped from the conjunction if $i = n{+}1$. We describe a solution to `lex` by an expression $(=)^{i-1}[<]$ and we identify it by the position i of the strict inequality $<$. The resulting sequence of inequalities is meant to hold between the respective pairs of variables from the two lists of the `lex` constraint. $[e]$ means that expression e is dropped if its position in the sequence is greater than n. An expression e^0 is also dropped.

We argue for completeness based on the following observations:

- There can be at most $n + 1$ solutions to a given `lex` constraint over lists of length n.
- The disjunction of all solutions of a `lex` constraint is logically equivalent to the constraint.
- Inequality constraints can be added to a `lex` constraint so that any subset of solutions is possible:
  - Imposing $x_i < y_i$ or $x_i \neq y_i$ means there can be a solution at position i, but not at any greater position, since equality is not possible anymore at position i.
  - Imposing $x_i = y_i$ or $x_i \geq y_i$ means there cannot be a solution at position i, but possibly at greater positions.
  - Imposing $x_i \leq y_i$ means there can be a solution at position i or any greater position.
  - Imposing $x_i > y_i$ means there cannot be a solution at position i or any greater position.
- Hence the smallest position that admits a solution is the first position i that admits $<$ (i.e. $<$, $\leq$, $\neq$, *true*) or where $i{=}n{+}1$, provided all previous positions admitted $=$ but not $<$ (i.e. $=$, $\geq$). If there is no such smallest position, then there is no solution.

Based on these observations, we distinguish two kinds of propagation.

Forward Propagation The new inequalities that we can propagate from a disjunction of the solutions of a given `lex` constraint together with some inequalities, i.e. all those built-in constraints that are implied, that must hold no matter which disjunct (solution) is chosen, are simply and only $(=)^{i-1}[\leq]$, where i is the first, smallest position of a solution.
Thus a complete implementation has to turn leading $\geq$ inequalities into equalities $=$, proceed over $=$ and impose $\leq$ on the first remaining other inequality. In our constraint solver implementation this is achieved by the propagation rule `l4` that imposes $\leq$ on any current first position and the recursive simplification rule `l3` that removes leading $=$.

Backward Propagation A special case arises if there is exactly one solution, in that case obviously the last inequality that we have to propagate can be made strict, $(=)^{i-1}[<]$. We have exactly one solution if there are no more solutions after the smallest position that admits a solution. This is the case if the smallest position is followed by a sequence of zero or more $=$ or $\geq$ constraints that is ended by $>$.
This special one-solution case is handled by the simplification rules `l5` and `l6`. Rule `l5` covers the case where $>$ holds for the second position, so $<$ must hold for the first position to ensure a solution. Rule `l6` allows one to reduce the other instances of the special case, where there is an arbitrary number of $=$ or $\geq$ constraints between the unique position for a solution and the $>$ inequality (that could also come from a $\geq$ being strenghened), to the situation in rule `l5`.

Note that the rules `l1` and `l2` are not needed for completeness of propagation, simply because they do not propagate anything except the trivial *true*. But the two rules are useful for garbage collection and `l2` is also needed for confluence.

8 Implementation Experiments

In the literature so far, the `lex` constraint has only been used for finite domains, so a comparison is only possible when this constraint system is chosen as built-in one. While the implementation of [8] seems not to be public domain, the implementation of [6] is included in the latest Sicstus Prolog releases.

We tested our CHR implementation of `lex` with the CHR library in Sicstus Prolog, while Tom Schrijvers was so kind to test it in SWI Prolog. The implementations are not incremental at this point, but can be made so by a tighter coupling. While some readers may be impressed with benchmark tables, we have omitted them for space reasons. The interested reader can find more detailed measures that would fit a paper online. The main test file with code and results for both Prolog implementations is available online (further test files are mentioned in that file):
`www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/lextest.pl`

For our tests we have used Sicstus 3.11 Prolog with standard settings running on a Suse Linux PC with medium overall work load. We compiled our code with the 'compactcode' option of Sicstus. In the implemented rules, we used a straightforward coupling between the CHR `lex` constraint utilizing the `chr` library of Sicstus and the finite domain built-in constraints of the Sicstus `clpfd` library, where we inspect domains in the guards to perform the necessary checks.

After some initial experiments we found that the propagation rule `l4` of the solver exhibits a non-linear (quadratic) behavior instead of a linear one. One reason is because every time a `lex` constraint suspends, all its variables in the constraint are scanned, while it would only be necessary to scan the first two variable pairs. We avoided this bottleneck by rewriting the propagation rule into a guarded simplification rule.

We compared our CHR implementation of `lex` with the built-in `lex_chain` constraint [6] of the Sicstus `clpfd` constraint library. This unary constraint takes a list of lists of domain variables with finite bounds or integers. The constraint holds if the lists are in ascending lexicographic order.

We considered lists up to 40000 elements, at around 50000 elements memory problems occurred. Garbage collection was never performed by the system. In our experiments, both lexicographic constraints showed a complete propagation behavior and linear time complexity. The number of rule applications in our solver is linear in the list length as calculated. Run-times were less than a second for the CHR `lex` constraint for simpler test cases. While forward propagation in CHR was just 3 times slower than built-in `lex_chain`, backward propagation proved to be 10–20 times slower, possibly because the recursive decomposition in the CHR solver generates many small `lex` constraints.

Tom Schrijvers has run the tests in his K.U. Leuven CHR system in an experimental version of SWI Prolog that will be included in the development version in early 2006. Due to compile-time suspension variable inference in that CHR implementation, scanning is improved so that the original propagation rule of the lexicographic constraint solver can be run without run-time penalty in linear time. In tests with up to 4000 list elements, a linear-time behavior was observed. Some additional time is spend in garbage collection.

9 Conclusions

Just six CHR rules correctly and efficiently specify and implement an incremental and concurrent, logical algorithm to maintain consistency of the lexicographic ordering constraint. Previous approaches presented algorithms for the lexicographic order constraint in pseudo-code that seems hard to analyse or use an automata formalism that seems hard to re-implement, while our solver program is simple, short, concise and directly executable. We have found a direct recursive decomposition of the problem that does not need additional constraints and performs all possible propagations. Moreover, our solver is independent of the underlying constraint system that provides inequalities between the elements of

the sequences to be compared lexicographically, and therefore our solver is not restricted to finite domains.

Our solver consists of three pairs of rules, the first two corresponding to base cases of the recursion (garbage collection), then two rules performing forward reasoning (recursive traversal and implied inequality), and finally two for backward reasoning, covering a special case when the lexicographic constraint has a unique solution. We have proven the rules to be confluent using our semi-automatic confluence checker. We showed logical correctness, completeness of constraint propagation and worst-case time complexity linear in the cost of handling the built-in inequality constraints.

We already know that, at least in theory, CHR can implement any algorithm in best-known space and time complexity [17], and many CHR constraint solvers including the `lex` constraint discussed here are practical proof that it is indeed possible. The remaining constant-factor slow-down observed in the implementation experiments is the price one currently has to pay for using a very high-level language as CHR in contrast to a low-level hard-wired implementation. Since the run-time increase is by a constant factor only, we can hope that compiler optimization will further close the performance gap.

Future work should consider extensions of the lexicographic ordering constraint that can be found in the recent literature, e.g. using it in chains or with a summation constraint, or simplifying `lex` constraints for symmetry breaking. As for the instantiations of the generic constraint solver to specific built-in constraint systems, several issues are open: To show that the finite domain instance maintains generalized arc consistency, to use other underlying built-in constraint systems such as linear polynomials or temporal constraints, and to give an implementation that does not rely on built-in constraints for inequality, but rather uses existing CHR solvers. Finally, a hard, challenging question is if and how rules such as the ones presented here can be derived automatically from inductive definitions.

Acknowledgements. The author would like to thank Marc Meister and Tom Schrijvers for discussions and help with implementation and testing, and the reviewers for their helpful comments.

References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *3rd International Conference on Principles and Practice of Constraint Programming*, LNCS 1330. Springer, 1997.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal, Special Issue on the 2nd International Conference on Principles and Practice of Constraint Programming*, 4(2):133–165, 1999.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Press, 1998.

4. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In M. Wallace, editor, *CP'2004, Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, Berlin, Heidelberg, New York, 2004. Springer.
5. M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, Swedish Institute of Computer Science, 2002.
6. M. Carlsson and N. Beldiceanu. From constraints to finite automata to filtering algorithms. In D. Schmidt, editor, *ESOP2004*, volume 2986 of *LNCS*, pages 94–108, Berlin, Heidelberg, New York, 2004. Springer.
7. H. Fargier, J. Lang, and T. Schiex. Selecting preferred solutions in fuzzy constraint satisfaction problems. In *1st European Congress on Fuzzy and Intelligent Technologies (EUFIT)*, 1993.
8. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. V. Hentenryck, editor, *CP'2002, Int. Conf. on Principles and Practice of Constraint Programming*, volume 2470 of *LNCS*, pages 93–108, Berlin, Heidelberg, New York, 2002. Springer.
9. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
10. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
11. T. Frühwirth. Logical rules for a lexicographic order constraint solver. In T. Schrijvers and T. Frühwirth, editors, *Second Workshop on Constraint Handling Rules, at ICLP 2005*, Sitges, Spain, October 2005.
12. T. Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In M. Gabbrielli and G. G., editors, *Logic Programming: 21st International Conference, ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 113–127. Springer, Oct. 2005.
13. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
14. B. Hnich, Z. Kiziltan, and T. Walsh. Combining symmetry breaking with other constraints: Lexicographic ordering with sums. In *AMAI 2004 Eighth International Symposium on Artificial Intelligence And Mathematics*, 2004.
15. T. Schrijvers, B. Demoen, G. Duck, P. Stuckey, and T. Frühwirth. Automatic implication checking for chr constraints. In *6th International Workshop on Rule-Based Programming*, volume 147 of *Electronic Notes in Theoretical Computer Science*, pages 93–111, Jan. 2006.
16. T. Schrijvers and T. Frühwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`, 2006.
17. J. Sneyers, T. Schrijvers, and B. Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.