Logical Rules for a Lexicographic Order Constraint Solver

Thom Frühwirth

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

Abstract. We give an executable specification of the global constraint of lexicographic order in Constraint Handling Rules (CHR) language. In contrast to previous approaches, the implementation is short and concise without giving up on linear time 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 also show completeness of constraint propagation, i.e. that all possible consequences of the constraint are generated by the implementation. Our algorithm is encoded by three pairs of rules, two corresponding to base cases, two performing the obvious traversal of the sequences to be compared and two covering a not so obvious special case when the lexicographic constraint has a unique solution.

1 Introduction

Lexicographic orderings are common in everyday life as e.g. the alphabetical order used in dictionaries and listings. 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. [15]) and earlier in modelling preferences among solutions (e.g. [8]).

Related Work. A natural question to ask is whether lexicographic orders can be implemented as constraints and what would be appropriate incremental propagation algorithms. There are two approaches to the problem, starting with [9] 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 [9] an algorithm based on two pointers that move along the elements of the sequences to be lexicographically ordered is given. The algorithm is not incremental and its 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 is either lengthy or requires subsequent translation into a different formalism. These specifications are 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 auxiliary constraints are needed. 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 show linear worst-case time complexity and that they propagate as much information (constraints) as is possible.

CHR [10, 14, 12] 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 developped 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 implementation in Section 3. Then, in separate sections, we discuss confluence, logical correctness, completeness and worst-case time complexity before we conclude. To the best of our knowledge, this is the first paper that discusses the implementation of a global constraint in CHR.

2 Preliminaries: Constraint Handling Rules

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [10, 14, 12]. Readers familiar with CHR can skip this section.

2.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

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

where *Name* is an optional, unique identifier of a rule, the l.h.s. *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the r.h.s. *body* B is a goal. A *goal (query, problem)* is a conjunction of built-in and CHR constraints. A trivial guard expression "`true |`" can be omitted from a rule.

Simpagation rules abbreviate simplification rules of the form

$$\textit{Name} \ @\ H \wedge H' \Leftrightarrow C \mid H \wedge B.$$

2.2 Declarative Semantics of CHR

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

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 $\bar{y}$ 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.

2.3 Operational Semantics of CHR

The operational semantics of CHR is given by a transition system, where computation states are goals, i.e. conjunctions of built-in and CHR constraints.

CHR rules are applied exhaustively, until a fixed-point is reached, to the initial state. 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, rule applications are restarted.

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 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 (in contrast to Prolog).

When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule. When a propagation rule is applied, the body of the rule is added to the goal without removing any constraints[1]. When a simpagation rule is applied, all constraints to the right of the backslash are replaced by the body of the rule.

3 A Lexicographic Order Constraint Solver

A lexicographic order allows to compare sequences by comparing the elements of the sequences proceeding from start to end of the sequences.

Definition 1. 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 $l_1 \preceq_{lex} l_2$ iff $n{=}0$ or $x_1{<}y_1$ or $x_1{=}y_1$ and $[x_2, \ldots, x_n] \preceq_{lex} [y_2, \ldots, y_n]$.

This inductive definition[2] can be directly modeled as a recursive predicate in Prolog (where sequences are represented by lists). The built-ins `<` and `=` are either tests or constraints. In the latter case, also non-ground lists can be handled.

```
[] lex [].
[X|L1] lex [Y|L2] :- X<Y.
[X|L1] lex [Y|L2] :- X=Y, L1 lex L2.
```

This executable specification in Prolog relies on backtracking search to find the applicable clauses. A smart compiler may remove clause choices for ground lists, but if the relationship between some variable pairs is not known, search cannot be avoided.

The logical reading of this Prolog program, Clark's completion [7], almost literally corresponds to the above definition, hence the specification is easily seen to be *logically correct.*

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

3.1 A First Implemention in CHR

We now rewrite this logical specification into CHR rules for the lexicographic order constraint. We assume that the lists to be compared are given, while their elements are variables or constants. Since the three disjuncts of the specification are mutually exclusive, we can easily turn each clause into a CHR simplification rule where the guards ensure the mutual exclusion.

[1] To avoid non-termination, a propagation rule is never applied a second time to the same constraints.

[2] All what follows will also work with sequences of different length and with strict lexicographic order, but one has to add a few more rules. We have chosen the same lenghts in order to be able to concentrate on the essential rules for propagation.

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

These 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. For example, the queries `[1] lex [2]`, `[X] lex [X]` and `[X] lex [Y], X<Y` will all reduce to `true`. To the queries `[X] lex [Y]`, `[X] lex [Y], X>=Y` and `[X] lex [Y], X>Y` no rules are applicable.

While the above program is correct, the rules do not propagate any constraints except the trivial `true`. We must do better than that. We can derive a common consequence of the last two clauses,

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

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

and implement it as a CHR propagation rule, where the built-in inequality constraint appears in the body of a rule and is thus enforced when the rule is applied[3]

```
l4 @ [X|L1] lex [Y|L2] ==> X=<Y.
```

For example, to the query `[R|Rs] lex [T|Ts], R`$\neq$`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 result is the remaining `R<T`.

However, the rules above are not sufficient, more propagations are possible. For example, consider `[R1,R2,R3] lex [T1,T2,T3], R2=T2, R3>T3`. The only way to satisfy this constraint is by asserting `R1<T1`, since the remaining elements cannot be ordered in the right way if `R1>=T1`. In order to perform such reasoning, we have to look forward, at the next level of recursion. If the recursive call fails, the base case for non-empty sequences must hold. If the recursive call proceeds to the next recursion, because the two variables are equal, we can ignore (i.e. remove) this pair of variables.

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

Admittedly, these last two rules require some insight.

Note that rules `l4` and `l5` are the only ones that directly impose a built-in constraint.

Our example, `[R1,R2,R3] lex [T1,T2,T3], R2=T2, R3>T3`, can now be handled. First, since `R2=T2`, rule `l6` is applicable, and its result is `R2=T2, R3>T3,`

[3] It is the responsibility of the compiler to generate appropriate code for built-in constraints depending on their use in guards and bodies.

[R1,R3] lex [T1,T3]. Now rule l5 is applied, and we arrive at R2=T2, R3>T3, R1<T1 as desired.

There are still situations that are not covered by the current set of rules. Just replace R2=T2 in the above example by R2>=T2. The same propagation should take place as before, but the two rules that we have added cannot be applied, their guards are too strict.

In these situations, the current pair of variables is followed by a sequence of variables which are pairwise in >= relation, which each could turn into a strict inequation later on. This can be covered by modifying rule l6 into a propagation rule and weaken its guard:

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

The six rules (l1 to l5 and l6') implement a complete constraint solver for the non-strict lexicographic order constraint for comparing sequences of the same, given length. The rules are obviously terminating. The recursions involve the lex-constraint only. Each recursive call proceeds with shorter lists (with one element less each). But the solver is not as efficient as it could be.

Time Complexity. To show that worst-case time complexity is not linear[4], we consider the longest sequences of computation steps (rule applications) caused by the propagation rule l6'. Assume [X,U1,...Un] lex [Y,V1,...Vn] with U1>=V1,...Un>=Vn. Repeated recursive application of the propagation rule l6' will result in n lex constraints [X,U1,...Un] lex [Y,V1,...Vn], [X,U2,...Un] lex [Y,V2,...Vn],...[X,Un] lex [Y,Vn] with $O(\mathtt{n}^2)$ occurrences of variables.

If one of the variable pairs Ui and Vi is turned into a strict inequality Ui>Vi, the pair X and Y will be constrained to X<Y. This in turn will reduce all pending lex constraints (that all start with this variable pair) to true.

On the other hand, if X=Y, all propagated lex-constraints will be reduced by this pair of variables, [U1,...Un] lex [V1,...Vn], [U2,...Un] lex [V2,...Vn],...[Un] lex [Vn]. All subsequent pairs Ui and Vi will be forced to be equivalent due to the propagation rule l4, and the reduction will continue until all variable pairs are removed from all lex constraints. The result is X=Y, U1=V1,...Un=Vn.

In the last computation scenario, we have to remove $O(\mathtt{n}^2)$ occurrences of variables. The propagation rule l6' turns an otherwise linear complexity into a quadratic one. The problem is the redundant information (repeated variables) contained in each of the lex-constraints generated by the propagation rule.

3.2 Improving Time Complexity

In order to regain linear time complexity, we try to replace the propagation rule l6' by an equally powerful simplification rule. Just changing ==> into <=>

[4] We assume a standard CHR implementation here. With specific optimizations, linear complexity is possible.

results in a rule that is logically incorrect and also results in non-confluence. A semantics-preserving translation of the propagation rule by adding the head to the body, `[X,U|L1] lex [Y,V|L2] <=> U>=V | [X,U|L1] lex [Y,V|L2], [X|L1] lex [Y|L2]`, leads to obvious non-termination. But it gives us a clue in that it shows the problem of the repated occurences of the subsequences `L1` and `L2`. As it turns out (see Section on Correctness), removing `L1` and `L2` from the added `lex` constraint preserves correctness and improves time complexity to linear.

Our lexicographic constraint solver consists now of the following 6 rules.

```
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 additional condition `L1=[_|_]` in the guard of rule `l6''` avoids non-termination in case `L1=[]`.

Our algorithm is encoded by three pairs of rules, the first two corresponding to base cases of the recursion, then two rules performing the obvious recursive traversal of the sequences to be compared and finally two covering a not so obvious special case when the lexicographic constraint has a unique solution.

As we will later see, the rules `l1` and `l2` are not needed for completeness of constraint propagation, but are useful for garbage collection, maintaining confluence and for detection of entailment (a constraint is entailed if it reduces to `true` without adding new constraints).

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 [11], 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 2. 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 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*, we consider 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 the critical pair is not joinable, we have found a counterexample for confluence of the program.

4.1 Confluence of the `lex`-Constraint

We used the confluence checker mentioned in [13]. 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.

For example, 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.
```

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 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).

Another interesting critical pair is between 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, so that the two states of the critical pair are the same. Hence it is joinable.

If rule `l2` or rule `l4` is dropped, the solver becomes non-confluent, while the other rules can be dropped without hurting confluence. Solver completeness is harmed if rules other than `l1` and `l2` are dropped. (This means that we could drop rule `l1` from the solver.)

5 Logical Correctness

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

$$
\begin{array}{rll}
 & ([] \preceq_{lex} []) & \\
X{<}Y \rightarrow & ([X|L1] \preceq_{lex} [Y|L2]) & \\
X{=}Y \rightarrow & ([X|L1] \preceq_{lex} [Y|L2] & \leftrightarrow L1 \preceq_{lex} L2) \\
 & ([X|L1] \preceq_{lex} [Y|L2] & \rightarrow X {\leq} Y) \\
U{>}V \rightarrow & ([X,U|L1] \preceq_{lex} [Y,V|L2] & \leftrightarrow X < Y) \\
(U{\geq}V \wedge L1{=}[a|b]) \rightarrow & ([X,U|L1] \preceq_{lex} [Y,V|L2] & \leftrightarrow \\
 & \multicolumn{2}{r}{([X,U] \preceq_{lex} [Y,V] \wedge [X|L1] \preceq_{lex} [Y|L2]))}
\end{array}
$$

It is easy to show that these formulas are logical consequences of the logical specification given by

$$
\begin{array}{ll}
l_1 \preceq_{lex} l_2 \leftrightarrow & (l_1{=}[] \wedge l_2{=}[]) \vee \\
 & (l_1{=}[x|l'_1] \wedge l_2{=}[y|l'_2] \wedge x{<}y) \vee \\
 & (l_1{=}[x|l'_1] \wedge l_2{=}[y|l'_2] \wedge x{=}y \wedge l'_1 \preceq_{lex} l'_2)
\end{array}
$$

As an example, we prove that the logical reading of rule `l6''` is a logical consequence of the logical specification of the `lex` constraint.

Logical Correctness Proof for Rule `l6''`*.* From the specification it follows

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

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

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

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

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 finite domains, linear polynomials, Booleans, admit these inequalities, and that the typical algorithms used (e.g. arc and path consistency, simplex) have different time complexities.

We therefore give our complexity result in the number of atomic built-in constraints that are checked and imposed, respectively.

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

To empty lists, only the rule `l1` is applicable. To lists with one or two elements, the rules `l2`, `l3`, `l4` and `l5` are applicable (but not rule `l6''`). The propagation rule `l4` can be applied exactly once to each `lex` constraint. Hence for lists of length less than or equal to 2, the number of rule applications is bounded by a constant (at most 5). Therefore the first recursive call of rule `l6''` involving a list of length 2 can only contribute a constant number of rule applications. Thus, for lists of length greater than 2, the number of rule applications is linear in the length of the list. Therefore the number of checks and additions of built-in inequality constraints is also linear in the length of the list.

In most constraint systems, the canonical form for constraints will require that all variables in a global constraint are different from each other (equivalent variables are equated by additional equality constraints). In that case, we can replace list length in the complexity result by the number of different variables in the list.

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 can be derived from the `lex` constraint.

A *solution* of a lexicographic order constraint $[x_1,\ldots,x_n]{\preceq_{lex}}[y_1,\ldots,y_n]$ is a conjunction of built-in inequality constraints that logically implies the order constraint. In practice, a solution can be obtained by unfolding the `lex` constraint away, as is done in the Prolog implementation of the constraint. It is easy to see that a solution will be 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 $i = n{+}1$. Hence there can be at most $n + 1$ solutions to a given `lex` constraint over lists of length n.

We describe such a solution by an expression $(=)^{i-1}[<]$. 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. With $x?y$ we will denote $x < y \vee x = y \vee x > y$, which always holds for total orders that we use. The symbol ; denotes a choice between alternatives.

Next we convince ourselves that there are `lex` constraints over lists of length n that have as solutions any subset of the $n + 1$ possible solutions. For example, given three positions $1 \leq i < j < k \leq n{+}1$, any `lex` constraints whose list elements satisfy the inequality constraints specified by the expression $(\geq)^i?(\geq)^{j-i}?(\geq)^{k-j}[(< \ ; \ ?(\geq)^* >)]$ will exactly have the three desired solutions $(=)^i <$, $(=)^j <$, and $(=)^k[<]$.

The expression allows to construct a `lex` constraint with the desired solutions. The idea is to leave those positions unconstrained where a solution should end (denoted by ? in the expression), and to constrain all other positions to $\geq$, so that a solution ending at that position is not possible, since it would require $<$. But equality $=$ is possible, which allows to pass over that position and enable solutions that go beyond that position. In addition, special care has to be taken with the last solution. In order to avoid a solution that equates all variable pairs, the last position for a solution either has to be constrained to $<$, so that it cannot be passed over, or a constraint on a later position has to be imposed so that passing over cannot lead to a solution, this is the case if at some later position we impose $>$ instead of $\geq$. This is described by the alternative $?(\geq)^* >$ at the end of the expression. Note that it implies that ? must be constrained to $<$ if we want to have a solution at that position.

Clearly, the disjunction of all solutions of a `lex` constraint is logically equivalent to the constraint. The new inequalities that we can propagate from such a disjunction, i.e. those built-in constraints that are implied, that must hold no matter which disjunct (solution) is chosen, are simply and only $(=)^i \leq$, where i is the position of the first solution. A special case arises if there is only one solution, in that case obviously $(=)^i[<]$. As discussed above, this case arises if the initial expression is $(\geq)^*[?(\geq)^* >]$.

These observations mean that a complete implementation has to turn leading $\geq$ inequalities into equalities $=$ and impose $\leq$ on the first 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 $=$. The special 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 to reduce the other instances of the special case, where there is an arbitrary number of $\geq$ constraints between the unique position for a solution and the $>$ inequality, to the situation in rule `l5`. (The rules `l1` and `l2` are not needed for completeness, but are useful simplifications to detect entailment.)

8 Conclusions

Just six CHR rules correctly and efficiently specify and implement an incremental, concurrent, logical algorithm to maintain consistency of the lexicographic ordering constraint. Previous approaches presented algorithms for the lexicographic order constraint in often lengthy pseudo-code that is hard to analyse, while our 6 rules are simple and directly executable. Moreover, our implementation is independent of the underlying constraint system, and therefore not restricted to finite domains. We have proven the rules to be confluent using our automatic confluence checker. We showed logical correctness, completeness of constraint propagation and linear worst-case time complexity.

Future work should give an implementation that does not rely on built-in constraints for inequality, but rather uses existing CHR solvers (e.g. for finite domains). Experimental evaluation would be next to find out the constant factors hidden in the time complexity. 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, could be considered. 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, Tom Schrijvers and the reviewers for their 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, Berlin, Heidelberg, New York, 1997. Springer.
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. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.
8. 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.
9. 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.
10. 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.
11. 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.
12. T. Frühwirth. CHR web-pages, www.informatik.uni-ulm.de/pm/ mitarbeiter/fruehwirth/chr.html, 2004.
13. T. Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In *International Conference on Logic Programming (ICLP)*, LNCS, Barcelona, Spain, 2005. Springer.
14. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.
15. 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.