Proving Termination of Constraint Solver Programs

Thom Frühwirth

Ludwig-Maximilians-Universität München
Oettingenstrasse 67, D-80538 Munich, Germany
fruehwir@informatik.uni-muenchen.de
www.informatik.uni-muenchen.de/~fruehwir/

Abstract. We adapt and extend existing approaches to termination in rule-based languages (logic programming and rewriting systems) to prove termination of actually implemented CHR constraint solvers.
CHR (Constraint Handling Rules) are a declarative language especially designed for writing constraint solvers. CHR are a concurrent constraint logic programming language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved.
The approach allows to prove termination of many constraint solvers, from Boolean and arithmetic to terminological and path-consistent constraints. Because of multi-heads, our termination orders must consider conjunctions, while atomic formulas suffice in usual approaches.
Our results indicate that in practice, proving termination for concurrent constraint logic programs may not be harder than for other classes of logic programming languages, contrary to what has been feared in the literature.

1 Introduction

We adapt and extend existing approaches to termination in rule-based languages (logic programming and rewriting systems) to prove termination of actually implemented CHR constraint solver programs.

CHR (Constraint handling rules) [Fru98,AFM99] are a high-level language especially designed for writing constraint solvers. CHR are a committed-choice concurrent constraint logic programming language consisting of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. CHR define both *simplification* of and *propagation* over user-defined constraints. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification. CHR have been used in dozens of projects worldwide to encode constraint solvers, including new domains such as terminological, spatial and temporal reasoning [Fru98] and new applications domains such as optimal placement of sender stations [FrBr00].

The study of termination of CHR programs is not only essential for reliable constraint solvers, termination is a prerequisite for analyzing and deciding confluence [Abd97,AFM99] and operational equivalence [AbFr99] of CHR programs.

Confluence guarantees that the result of a computation will always be the same, no matter which of the applicable rules are applied.

In logic programming in general, a termination problem can only occur if recursion is involved. Once recursion is present, the problem is almost at once undecidable. There is a fair amount of work on sufficient conditions ensuring termination of (pure) logic programs [dSD94], which started about a decade ago. The basic idea is to prove that in each rule, the head atom is strictly larger than every atom occurring in the body of the rule.

Typically, the necessary well-founded orders are adopted from term rewriting systems (TRS). A commonly used order is called *polynomial interpretation* which is known in TRS since more than twenty years [Der87,BaNi98]. The idea is to map terms and atoms to natural numbers. Instances of this mapping are also called measure function, norm, ranking or level mapping. To ensure well-foundedness, programs and queries usually have to be well-moded (and well-typed) or queries sufficiently known.

The main line of work in termination of logic programs is considered to be from Apt, Bezem and Pedreschi [ApPe90,Bez93]. Both programs and goals are characterized in terms of *level mappings*, a function from ground atoms to natural numbers. A logic program is *recurrent* if for every ground instance of each rule, the *level* of the head atom is higher than the level of each body atom. A goal is *bounded (rigid)* if for every (ground) instance of each atom in the goal there is a maximum level which is not exceeded. Successive work of the authors refined this approach: Local variables and the specific left-to-right SLD resolution of Prolog are taken into account. A program is *acceptable* if for every ground instance of each rule the level of the head atom is higher than the level of each body atom whenever it is not in the model of the program and all the body atoms on the right are in the model. The model of a program is characterized by suitable *interargument relations* that must hold on the atoms. The notion of bounded goals is extended as well to take the model into account.

There are only a few recent papers on termination for constraint logic programs [CMM95,Mes96,Rug97], logic programs with coroutining [Nai92,MaTe95] and concurrent logic programs [Plu92,KKS97]. [Rug97,Mes96,MaTe95,Plu92] embark on level mappings. The theoretical work [CMM95] provides necessary and sufficient conditions for termination based on dataflow graphs, the practical work [Nai92] discusses informally how terminating procedures can be combined ensuring overall termination, and [KKS97] can use techniques from TRS directly since they translate GHC programs into TRS.

To the best of our knowledge, there is no work yet on proving termination of concurrent constraint logic programs and of constraint solver implementations.

In the literature it is generally agreed that the issue of termination for concurrent constraint languages is even harder than for other logic programs, since programs with constraints do not go well with the idea of modes and well-modedness, and since programs with coroutining or concurrency do not have a statically fixed search and selection rule.

The following example illustrates the behavior of committed-choice languages with respect to delaying and termination.

Example 1. Consider a CHR constraint characterizing even numbers. We use Prolog syntax, where Variables start with upper case letters, and function and predicate symbols with lower case letters. Assume that numbers are expressed in successor notation and that = means syntactical equality. The constraint may be defined by the single rule:

```
even(X) <=> X=s(Y) | Y=s(Z), even(Z).
```

The rule says that if the argument `X` to `even` is the successor of some number `Y`, then the predecessor of this number `Z` must be even in order to ensure that the initial number `X` is even. The query `even(N)` delays. The query `even(f(N))` delays as well. To the query `even(s(N))` the rule is applicable, the answer is `N=s(N1), even(N1)`.

It was already discussed in detail in [Nai92] that the conjunction of two query atoms, that both terminate on its own, need not terminate. Here, the query `even(N), even(s(N))` will not terminate. It leads to `even(N), N=s(N1), even(N1)`, which is equivalent to `even(s(N1)), even(N1)`, which is just a variant of the initial query.

For CHR, we not only have concurrency and constraints, but also propagation rules and multiple heads to consider. Thus we cannot hope to give a definitive or final answer concerning termination at this point in time. In this paper we rather concentrate on ensuring termination in practice, in existing constraint solvers written in CHR.

Overview of the Paper. We will first give syntax and semantics for CHR. In the next section, we introduce useful termination orders for CHR. Then we prove termination of actually implemented CHR constraint solvers ranging from Boolean and arithmetic to terminological and path-consistent constraints. Finally, we summarize the achievements of the current approach and discuss future work.

2 Syntax and Semantics

In this section we give syntax and simple semantics for CHR, for more detailed semantics see [Abd97,AFM99]. We assume some familiarity with (concurrent) constraint (logic) programming [vHSD92,JaMa94,FrAb97,MaSt98,CDJK99].

A *constraint* is a predicate (atomic formula) in first-order logic. We distinguish between *built-in (predefined) constraints* and *CHR (user-defined) constraints*. Built-in constraints are those handled by a predefined, given constraint solver. CHR constraints are those defined by a CHR program.

The *syntax* of CHR is defined by EBNF grammar rules and is reminiscent of Prolog and GHC. Upper case letters stand for conjunctions of constraints.

Definition 1. A CHR program is a finite set of CHR. There are two kinds of CHR. A *simplification CHR* is of the form

```
[N '@'] H '<=>' [G '|'] B.
```

and a *propaggation CHR* is of the form

```
[N '@'] H '==>' [G '|'] B.
```

where the rule has an optional name `N`, the multi-head `H` is a conjunction of CHR constraints. The optional guard `G` is a conjunction of built-in constraints. The body `B` is a conjunction of built-in and CHR constraints. As in Prolog syntax, a conjunction is a sequence of conjuncts separated by commas.

The *declarative semantics* of a CHR program P is a conjunction of universally quantified logical formulas (one for each rule) and a consistent built–in *constraint theory* CT which determines the meaning of the built–in constraints appearing in the program. The theory CT is expected to include an syntactical equality constraint `=` and the basic trivial constraints `true` and `false`. The declarative reading of a rule relates heads and body provided the guard is true. A simplification rule means that the heads are true if and only if the body is true. A propagation rule means that the body is true if the heads are true.

The *operational semantics* of CHR programs is given by a state transition system. With *computation steps (transitions, reductions)* one can proceed from one state to the next. A *computation* is a sequence of computation steps.

Definition 2. A *state (or: goal)* is a conjunction of built-in and CHR constraints. An *initial state (or: query)* is an arbitrary state. In a *final state (or: answer)* either the built-in constraints are inconsistent or no computation step is possible anymore.

Definition 3. Let P be a CHR program for the CHR constraints and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows. All variables occurring in states stand for conjunctions of constraints. $\bar{x}$ denotes the variables occurring in the rule chosen from P.

Simplify
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge D$
if $(H$ `<=>` $G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$
Propagate
$H' \wedge D \longmapsto (H = H') \wedge G \wedge B \wedge H' \wedge D$
if $(H$ `==>` $G \mid B)$ in P and $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$

By equating two atomic constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$ syntactically, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. By $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$ we mean $(H_1 = H'_1) \wedge \ldots \wedge (H_n = H'_n)$. Conjuncts can be permuted since conjunction is assumed to be associative and commutative.

When we use a rule from the program, we will rename its variables using new symbols. A rule is *applicable* to CHR constraints H' whenever these atomic constraints match (are an instance of) the head atoms H of the rule and the guard G is entailed (implied) by the built-in constraint store. The matching is the effect of the existential quantification in $\exists \bar{x}(H = H')$ [Mah87]. Matching and entailment checks are performed by the constraint solver for the built-in constraints. Any of the applicable rules can be applied, but it is a committed choice, it cannot be undone.

If an applicable simplification rule `(`H `<=>` G `|` B`)` is applied to the CHR constraints H', the **Simplify** transition removes H' from the state, adds B and also adds the equation $H = H'$ and the guard G to the state. If a propagation rule `(`H `==>` G `|` B`)` is applied to H', the **Propagate** transition adds B and also adds the equation $H = H'$ and the guard G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints. A more complex operational semantics that addresses these issues can be found in [Abd97]. Details on how to efficiently implement the operational semantics given here can be found in [HoFr00]. The examples in Section 4 will help to clarify the operational semantics.

3 CHR Termination Orders

To prove termination of CHR computations, we rely on polynomial interpretations, where the rank of a term or atom is defined by a linear positive combination of the rankings of its arguments. In the following we define a scheme for a class of rankings that we will use in the sequel to prove termination of constraint solvers written in CHR. The basic definitions follow [dSD94,BaNi98].

Definition 4. Let f be a function or predicate symbol of arity n ($n \geq 0$). A *CHR ranking (function)* defines the rank of a term or atom $f(t_1, \ldots, t_n)$ as a natural number:

$$rank(f(t_1, \ldots, t_n)) = a_0^f + a_1^f * rank(t_1) + \ldots + a_n^f * rank(t_n)$$

where the a_i^f are natural numbers. For each variable X we impose $rank(X) \geq 0$.

This definition implies that $rank(t) \geq 0$ for all rankings in our scheme and for all terms and atoms t.

Instances of the ranking scheme *rank* specify the function and predicate symbols and the values of the coefficients a_i^f.

Example 2. The size of a term can be expressed in this scheme by:

$$size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots + size(t_n)$$

For example, the size of the term `f(a,g(b,c))` is 5. The size of `f(a,X)` is $2 + size(X)$ with $rank(X) \geq 0$ when no additional constraints are introduced for ranks of variables. This allows us to conclude that the term `f(g(X),X)` is larger in size than `f(a,X)` ($2 + 2 * size(X) \geq 2 + size(X)$), no matter what term X stands for.

An expression $rank(s) \odot rank(t)$ is called an *order constraint*, where $\odot \in \{<, \leq, =, \neq, \geq, >\}$. To avoid clutter, we also write $s \succ t$ for $rank(s) > rank(t)$. The notion of order constraints will be used to formalize interargument relations, i.e. relations between the ranks of arguments of constraints that are needed to prove termination.

Two important properties of termination orders are well-foundedness and stability.

Definition 5. An order is *well-founded* if there are no infinite decreasing chains $r_1, \ldots, r_n \ldots$ such that $r_i \succ r_{i+1}$ for all $i \geq 1$.

Definition 6. An order $\succ$ has the *stability property* if it is closed under substitutions:

$$s \succ t \rightarrow \sigma(s) \succ \sigma(t) \text{ for all terms } s \text{ and } t \text{ and for all substitutions } \sigma,$$

where t contains only variables that also appear in s.

Linear polynomial orders (definable by our ranking function scheme) are known to be well-founded and closed under substitutions [BaNi98].

Induced Orders on Sequences and Multi-sets

If a ranking does not suffice to prove termination, it can be used to induce an order on sequences of finite length (tuples) and multi-sets [Der87]. In this paper, for one constraint solver we have to rely on multi-sets to prove its termination (see Section 4.3).

Definition 7. The *lexicographic order* on sequences is defined by:

$$(s_1, \ldots, s_n) \succ_l (t_1, \ldots, t_m)$$

iff there exists i with $(1 \leq i \leq n)$ such that $s_i \succ t_i$ or $i > m$ and for all j with $(1 \leq j < i)$ it holds that $s_j = t_j$.

If the sequences have different size, this only matters if the shorter sequence is a prefix of the longer one. In that case, the longer sequence is larger.

Definition 8. The *multi-set order* on multi-sets is defined by:

$$\{s_1, \ldots, s_n\} \succ_m \{t_1, \ldots, t_m\}$$

iff there exist $i \in \{1, \ldots, n\}$ and $1 \leq j_1 < \ldots < j_k \leq m$ with $0 \leq k$ such that $s_i \succ t_{j_1}, \ldots, s_i \succ t_{j_k}$ and $S \succ_m T$ or $S =_m T$ where $S = \{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n\}$ and $T = \{t_1, \ldots, t_m\} - \{t_{j_1}, \ldots, t_{j_k}\}$.

In words: The smaller multi-set is obtained from the larger one by removing a nonempty subset and adding only elements which are smaller than some element in the subset. Or: Every element in the smaller multi-set must be smaller or equal than one in the larger multi-set, and there must be at least one element that is strictly smaller. Even though the definition is contrived, there is a simple way to compare multi-sets for total orders: Sort the multi-sets into descending order and compare the resulting sequences lexicographically.

Sequences and multi-set orders induced by linear polynomial orders are also well-founded and closed under substitutions [BaNi98]. When such induced orders are used, the ranking function has to be lifted accordingly by introducing auxiliary functions that yield sequences and multi-sets respectively from ranks (see Section 4.3 for an example).

4 Proving Termination of Constraint Solvers

We are interested in termination of actually implemented CHR constraint solver programs. We want to prove termination under any scheduling of rule applications (independent from the search and selection rule). We also want to make sure that a conjunction of terminating queries is itself a terminating query. This means that we embark on a rather strict notion of termination.

Definition 9. A CHR program P is called *terminating for a class of queries*, if there are no infinite sequences of computation steps using rules from P starting from a query in the class.

Roughly, a CHR program can be proved to terminate if we can prove that in each rule, the rank of the head is strictly larger than the rank of the body.

A ranking for a CHR program will have to define the ranks of CHR and built-in constraints. In extension of usual approaches, we also have to define the rank of a conjunction of constraints, since CHR are multi-headed. We will define the rank of any built-in constraint to be the smallest element in the order (i.e. 0 or $\{\}$ for multi-sets), since we assume that they always terminate. The rank of a conjunction should reflect that conjunctions of CHR constraints are associative and commutative, but not idempotent. Thus obvious choices are $+$, and $\cup$ for multi-sets, repectively.

A built-in constraint may imply order constraints between the ranks of its arguments (interargument relations), e.g. $s{=}t \rightarrow \mathit{rank}(s) = \mathit{rank}(t)$. We assume these order constraints are given and known to be correct.

In this paper, we formalize a termination condition for simplification rules. We currently cannot deal with propagation rules in their generality, rather we will deal with them in a solver-dependent way.

Definition 10. The *ranking condition of a simplification rule* `H <=> G | B` is the formula

$$\forall\ (RC(G,B) \rightarrow H \succ B),$$

where $RC(G,B)$ is a conjunction of order constraints implied by the built-in constraints in the guard and body of the rule.

Since rankings are based on linear polynomial orders, $H \succ B$ does not universally hold if B contains local variables not occurring in H, except if the order constraints $RC(G,B)$ imply an appropriate relationship between the variables.

The intuition behind the definition of a ranking condition is that the built-in constraints in the rule will imply order constraints $RC(G,B)$ that can help us

to establish that $H \succ B$. There is no need in $RC(G, B)$ to distinguish between built-in constraints from the guard and from the body, even though they avoid non-termination for different reasons: If the constraints in the guard do not hold, the rule is not applicable, and neither is any instance of it. If the built-in constraints in the body do not hold, the application of the rule leads to an inconsistent, thus final state.

To prove termination, goals have to be sufficiently known.

Definition 11. A goal A is *bounded* if the rank of any instance of A is bounded from above by a constant k.

Obviously, the rank of a ground (variable-free) term is always bounded. Intuitively, in bounded goals, variables only appear in positions which are ignored by the ranking. The use of well-modedness instead of boundedness is not helpful in programs that define constraints, which should allow for arbitrary modes by definition, see Example 1. Obviously, boundedness generalizes the notion of modes.

The following two analogous theorems tell us how to prove CHR termination.

Theorem 1. Given a CHR program P without propagation rules and a ranking where

$$rank((A \wedge B)) = rank(A) + rank(B)$$

for any two goals A and B. If the ranking condition holds for each rule in P, then P is terminating for all bounded goals.

Proof. Consider a state $H' \wedge D$. Applying the rule (H <=> $G \mid B$) will lead to the state $(H = H') \wedge G \wedge B \wedge D$.

We have to show that $rank(H' \wedge D) > rank((H = H') \wedge G \wedge B \wedge D)$ and that the ranks of all states in a computation are bounded.

We know that $rank(G) = 0$, $rank(H = H') = 0$, since 0 is the smallest element in our polynomial order, and that $(H = H') \rightarrow rank(H) = rank(H')$.

Since $RC(G, B) \rightarrow rank(H) > rank(B)$, we have that

$$rank(H' \wedge D) = rank(H') + rank(D) = rank(H) + rank(D) >$$
$$0 + 0 + rank(B) + rank(D) = rank(((H = H') \wedge G \wedge B \wedge D)).$$

To show that the ranks of all states are bounded, note the following: Any ranking is well-founded and has the stability property. Since goals are bounded, the rank of a state is bounded. Due to the ranking condition, the boundedness of the source state is propagated to target state, i.e. given a bounded state $H' \wedge D$, the application of any simplification rule will lead to a state that is bounded again. Thus no infinite computations are possible, hence P is terminating.

The second Theorem is analogous to the first one, except that we consider multi-set orders.

Theorem 2. Given a CHR program P without propagation rules and a multi-set order defined by a function *mrank* which is induced by a polynomial ranking *rank*, and such that

$$mrank((A \wedge B)) = mrank(A) \cup mrank(B),$$

where A and B are goals and $\cup$ denotes multi-set union. If the ranking condition holds for each rule in P, then P is terminating for all bounded goals.

Proof. Consider a state $H' \wedge D$. Applying the rule (H `<=>` $G \mid B$) will lead to the state $(H = H') \wedge G \wedge B \wedge D$.

We have to show that $mrank(H' \wedge D) \supset mrank((H = H') \wedge G \wedge B \wedge D)$ and that the ranks of all states in a computation are bounded.

We know that $mrank(G) = \{\}$, $mrank(H = H') = \{\}$, since $\{\}$ is the smallest element in the multi-set order, and that $(H = H') \rightarrow mrank(H) = mrank(H')$.

Since $RC(G, B) \rightarrow mrank(H) \supset mrank(B)$, we have that

$$mrank(H' \wedge D) = mrank(H') \cup mrank(D) = mrank(H) \cup mrank(D) \supset$$
$$\{\} \cup \{\} \cup mrank(B) \cup mrank(D) = mrank(((H = H') \wedge G \wedge B \wedge D)).$$

To show that the ranks of all states are bounded, note the following: Any multi-set order induced by a polynomial ranking is well-founded and has the stability property. Since goals are bounded, the rank of a goal is bounded. Since there is only a finite number of goals in a state, the multi-set of its ranks is finite. Due to the ranking condition, the boundedness of the source state is propagated to target state. Thus no infinite computations are possible, hence P is terminating.

We are now ready to prove termination of actually implemented CHR constraint solvers ranging from Boolean and arithmetic to terminological and path-consistent constraints. For details on the constraint solvers analyzed here see [Fru98] and the CHR web pages:
`www.pst.informatik.uni-muenchen.de/~fruehwir/chr-intro.html`

4.1 Boolean Algebra, Propositional Logic

The domain of Boolean constraints includes the constants `0` for falsity, `1` for truth and the usual logical connectives of propositional logic, e.g. `and, or, neg, imp, exor`, modeled here as relations. Syntactical equality `=` is a built-in constraint. As a first, simple, but nevertheless useful example for a constraint solver, we can define an `and` constraint using value propagation, a special case of arc consistency:

```
and(X,Y,Z) <=> X=0 | Z=0.
and(X,Y,Z) <=> Y=0 | Z=0.
and(X,Y,Z) <=> X=1 | Y=Z.
and(X,Y,Z) <=> Y=1 | X=Z.
and(X,Y,Z) <=> Z=1 | X=1,Y=1.
and(X,Y,Z) <=> X=Y | Y=Z.
```

For example, the first rule says that the constraint `and(X,Y,Z)`, when it is known that the first input argument `X` is 0, can be reduced to asserting that the output `Z` must be 0. Hence the query `and(X,Y,Z),X=0` will result in `X=0, Z=0`.

The above rules terminate, since the CHR constraints `and` is not recursive. Any ranking that maps `and` to a positive number suffices to show this. The same holds for the other connectives.

In general, in termination proofs we can ignore rules whose body contains only built-in constraints.

A constraint solver is *complete* if can always reduce inconsistent CHR constraints to `false`. To achieve completeness for Boolean constraints as defined here, search must be employed by trying out values for the variables. In general, one is happy with incomplete solvers, because they have polynomial time-complexity as opposed to the exponential complexity of complete algorithms. Completeness has nothing to do with termination, but is mentioned in this paper to characterize the constraint solvers.

Boolean Cardinality

The cardinality constraint combinator was introduced in the CLP language cc(FD) [vHSD92] for finite domains. We adapted it for Boolean variables. The Boolean cardinality constraint `#(L,U,BL,N)` holds if between `L` and `U` Boolean variables in the list `BL` are equal to `1`. `N` is the length of the list `BL`. Boolean cardinality can express e.g. negation `#(0,0,[C],1)`, exclusive or `#(1,1,[C1,C2],2)`, conjunction `#(N,N,[C1,...,Cn],N)`, and disjunction `#(1,N,[C1,...,Cn],N)`.

In the following code, all constraints except cardinality `#` are built-in.

```
% trivial, positive and negative satisfaction
triv_sat@ #(L,U,BL,N) <=> L=<0,N=<U | true.
pos_sat @ #(L,U,BL,N) <=> L=N | all(1,BL).
neg_sat @ #(L,U,BL,N) <=> U=0 | all(0,BL).

% positive and negative reduction
pos_red @ #(L,U,BL,N) <=> delete(1,BL,BL1)| 0<U,#(L-1,U-1,BL1,N-1).
neg_red @ #(L,U,BL,N) <=> delete(0,BL,BL1)| L<N,#(L,U,BL1,N-1).
```

`all(T,L)` binds all elements of the list `L` to `T`. `delete(X,L,L1)` deletes the element `X` from the list `L` resulting in the list `L1`. When `delete/3` is used in the guard, it will only succeed if the element to be removed actually occurs in the list. E.g. `delete(1,BL,BL1)` will delay if it tries to bind a variable in `BL` to `1`. It will only succeed if there actually is a `1` in the list. It will fail, if all elements of the list are zeros.

Termination. The rules are still simple (single-headed simplification rules), but some are recursive. Since the cardinality constraint is either simplified into a built-in constraint (satisfaction rules) or reduced to a cardinality with a shorter list (reduction rules), this implementation terminates.

More formally, our termination proof is based on the length of the list:

$$rank(\#(L, U, BL, N)) = length(BL)$$

The length of a list can be expressed in our ranking scheme by:

$$length([]) = 0$$
$$length([X|L]) = 1 + length(L)$$

For example, the length of `[a,b,c,d]` is 4, the length of `[a|L]` is $1 + length(L)$ with $length(L) \geq 0$.

Remember that the rank of built-in constraints is always 0, but that they may imply order constraints. This is the case for `delete/3`:

$$delete(X, L, L1) \rightarrow length(L) = length(L1) + 1$$

Finally, the rank of a conjunction is the sum of the ranks of its conjuncts:

$$rank((A \wedge B)) = rank(A) + rank(B)$$

The interesting case for termination are the two reduction rules, because they are recursive. From the rule

```
pos_red @ #(L,U,BL,N) <=> delete(1,BL,BL1)| 0<U,#(L-1,U-1,BL1,N-1).
```

we get to prove

$$length(BL) = length(BL1) + 1 \;\rightarrow\; length(BL) > length(BL1).$$

The ranking condition holds, and in the same way we prove termination for the `neg_red` rule.

Due to the ranking, a goal consisting of built-in and cardinality constraints is bounded if the lengths of the lists in the cardinality constraints are known, i.e. if the lists are closed. If a list was open(-ended), there could be producers of an infinite list, and then the associated cardinality constraint would not necessarily terminate.

4.2 Terminological Reasoning

Terminological formalisms (aka description logics) [BaHa91] are used to represent the terminological knowledge of a particular problem domain on an abstract logical level. To describe this kind of knowledge, one starts with atomic concepts and roles, and then defines new concepts and their relationship in terms of existing concepts and roles. Concepts can be considered as unary relations similar to types. Roles correspond to binary relations over objects. Although there is an established notation for terminologies, we use a more verbose syntax to help readers not familiar with the formalism.

Definition 12. *Concept terms* are defined inductively: Every *concept (name)* c is a concept term. If s and t are concept terms and r is a *role (name)*, then the following expressions are also concept terms:

s `and` t (conjunction), s `or` t (disjunction), `nota` s (complement),
`every` r `is` s (value restriction), `some` r `is` s (exists-in restriction).

Objects are constants or variables. Let a, b be objects. Then $a : s$ is a *membership assertion* and $(a, b) : r$ is a *role-filler assertion*. An *A-box* is a conjunction of membership and role-filler assertions.

Definition 13. A *terminology (T-box)* consists of a finite set of *concept definitions*

c `isa` s,

where c is a newly introduced concept name and s is a concept term.

Since the concept c is new, it cannot be defined in terms of itself, i.e. concept definitions are acyclic (non-recursive). This also implies that there are concepts without definition, they are called *primitive*.

The CHR constraint solver for terminologies encodes the T-box by rules and the A-box as CHR constraints, since we want to solve problems over a given terminology (T-box). A similar solver is described in [FrHa95].

The consistency test of A-boxes simplifies and propagates the assertions in the A-box to make the knowledge more explicit and looks for obvious contradictions ("clashes") such as `X : device`, `X : nota device`. This is expressed by the rule:

```
I : nota S, I : S  <=> false.
```

The following simplification CHR show how the complement operator `nota` can be pushed towards to the leaves of a concept term:

```
I : nota nota S  <=>  I : S.
I : nota (S or T)  <=>  I : nota S and nota T.
I : nota (S and T)  <=>  I : (nota S or nota T).
I : nota (every R is S)  <=>  I : some R is nota S.
I : nota some R is S  <=>  I : every R is nota S.
```

An exists-in restriction generates a new variable that serves as a "witness" for the restriction:

```
I : some R is S  <=>  (I,J) : R, J : S.
```

A value restriction has to be propagated to all role fillers:

```
I : every R is S, (I,J) : R  ==>  J : S.
```

The unfolding rules replace concept names by their definitions. For each concept definition `C isa S` two rules are introduced:

```
I : C  <=>  I : S.
I : nota C  <=>  I : nota S.
```

The conjunction rule generates two new, smaller assertions:

```
I : S and T  <=>  I : S, I : T.
```

The rules simplify terminological constraints until a normal form is reached. The normal form is either `false` (inconsistent) or contains constraints of the form `I : C`, `I : nota C`, `I : S or T`, `I : every R is S` and `(I,J) : R`, where `C` is a primitive concept name. There are no clashes and the value restriction has been propagated to every object. To achieve completeness, search must be employed. This is done by splitting `I : S or T` into two cases, `I : S` and `I: T`.

Termination. The only CHR constraints that are rewritten by the rules are membership assertions. Since there are no guards, to show termination it therefore suffices to show that in each rule, the membership assertions in the body are strictly smaller than the ones in the head.

To prove termination we order concept terms by the following ranking:

$rank((A \wedge B)) = rank(A) + rank(B)$
$rank((I, J) : r) = 0$
$rank(I : s) = rank(s)$
$rank(nota\ s) = 2 * rank(s)$
$rank(c) = 1 + rank(s)$ if (`c isa s`) exists
$rank(f(t_1, \ldots, t_n)) = 1 + rank(t_1) + \ldots + rank(t_n)$ otherwise

The ranking above is well-founded, since concept definitions `c isa s` are acyclic and finite by definition. From the ranking we can see that goals are bounded if the ranks of all concept terms (like s and c) are known. Since concept terms are ground (variable-free) and finite by definition, their ranks can always be computed.

The propagation rule for value restrictions needs closer consideration. Note three things: First, the rank of its body is strictly smaller than the rank of its head. Second, since a propagation rule is applicable only at most once to the same constraints, it can only be applied a finite number of times to a finite conjunction of constraints. Third, the ranking is well-founded and goals are bounded. For these reasons, the propagation rule can only generate a finite number of smaller and smaller membership assertions.

4.3 Linear Polynomial Equations

For solving linear polynomial equations, a minimalistic but powerful variant of variable elimination [Imb95] is employed in the available CHR constraint solvers.

Definition 14. A *linear polynomial equation* is of the form $p + b = 0$ where b is a constant and the polynomial p is the sum of monomials of the form $a_i * x_i$ with coefficient $a_i \neq 0$ and x_i is a variable. Constants and coefficients are numbers. Variables are totally ordered by $\succ$. In an equation $a_1 * x_1 + \ldots + a_n * x_n + b = 0$, variables appear in strictly descending order.

In constraint logic programming, constraints are added incrementally. Therefore we cannot eliminate a variable in *all* other equations at once, but rather

consider the other equations one by one. A simple normal form can exhibit inconsistency: It suffices if the left-most variable of each equation is the only left-most occurrence of this variable. Therefore the two rules below implement a complete and rather efficient solver for linear equations over both floating point numbers (to approximate real numbers) and rational numbers. In the implementation, we write `eq` for equality on polynomials.

```
empty @ B eq 0 <=> number(B) | B=0.

eliminate @
A1*X+P1 eq 0, A2*X+P2 eq 0 <=>
   compute(P2-P1*A2/A1,P3),
   A1*X+P1 eq 0, P3 eq 0.
```

The `empty` rule says that if the polynomial contains no more variables, the constant `B` must be (approximate to) zero. The `eliminate` rule performs variable elimination. It takes two equations that start with the same variable. The first equation is left unchanged, it is used to eliminate the occurrence of the common variable in the second equation. The auxiliary built-in constraint `compute` simplifies a polynomial arithmetic expression into a new polynomial. No variable is made explicit, i.e. no pivoting is performed. Any two equations with the same first variable can react with each other. Therefore, the solver is highly concurrent and distributed.

The solver can be extended by a few rules to create explicit variable bindings, to make implicit equalities between variables explicit, to deal with inequations using slack variables or Fourier's algorithm.

Termination. Since in termination proofs we can ignore rules whose body contains only built-in constraints, we are only concerned with the *eliminate* rule here. To prove its termination we order the equations by the following ranking *mrank* that uses a multi-set order $\succ_m$ on the variables occurring in goals. The order is induced by the order on ranked variables $\succ$, the ranking function *rank* itself is not defined for any predicate or function symbols.

$$mrank((A \wedge B)) = mrank(A) \cup mrank(B)$$
$$mrank(P \ eq \ 0) = mrank(P)$$
$$mrank(A) = \{\} \text{ if } A \text{ is a built-in constraint}$$
$$mrank(T) = \{rank(V) \mid V \text{ is a variable in } T\} \text{ if } T \text{ is a term}$$

$$a_1 * X_1 + \ldots + a_n * X_n + b \ eq \ 0 \rightarrow X_i \succ X_j \text{ for all } n \geq i > j \geq 1 \quad (1)$$
$$compute(E, P) \rightarrow mrank(E) \supseteq mrank(P) \quad (2)$$

The order constraint (1) says that the monomials in the equations are ordered by their variables. The order constraints (2) says that the built-in constraint `compute` does not introduce new variables, but may eliminate occurrences of some.

For better readability, we will now just write the polynomial P instead of $mrank(P)$ and the variable V instead of $rank(V)$. From the `eliminate` rule we get that the head rank, the multi-set $(\{X\} \cup P_1 \cup \{X\} \cup P2)$, must be strictly larger than the body rank $(\{X\} \cup P_1 \cup P3)$. From the order constraint (2) we can derive that $P_2 \cup P_1 \supseteq P3$. Hence the body rank multi-set contains only variables from the head rank multi-set. Due to (1) we know that the variable X does not occur in P_1, P_2 and P_3, and that it comes before all other variables in P_1, P_2 and P_3 in the variable order. Therefore the head rank multi-set is strictly larger in the multi-set order than the body rank multi-set, because in the former X occurs twice and in the latter X occurs only once.

The order of the monomials by variables in the polynomial equations corresponds to an implementation of the multi-set order by a lexicographic order as mentioned at the end of section 3.

4.4 Path Consistency

In this section we analyze termination of constraint solvers that implement instances of the classical artificial intelligence algorithm of path consistency to simplify constraint satisfaction problems [MaFr85].

Definition 15. A *binary constraint network* consists of a set of variables and a set of (disjunctive) binary constraints between them. The network can be represented by a *directed constraint graph*, where the nodes denote variables and the arcs are labeled by binary constraints. Logically, a network is a conjunction of binary constraints.

Definition 16. A *disjunctive binary constraint* c_{XY} between two variables X and Y, also written $X\ \{r_1, \ldots, r_n\}\ Y$, is a finite disjunction $(X\ r_1\ Y) \vee \ldots \vee (X\ r_n\ Y)$, where each r_i is a relation that is applicable to X and Y. The r_i are called *primitive constraints*. The *converse* of a primitive constraint r between X and Y is the primitive constraint s that holds between Y and X as a consequence.

For example, $A\ \{<\}\ B, A\ \{<, >\}\ B, A\ \{<, =, >\}\ B$ are disjunctive binary constraints c_{AB} between A and B. $A\ \{<\}\ B$ is the same as $A < B$, $A\ \{<, >\}\ B$ is the same as $A \neq B$. Finally, $A\ \{<, =, >\}\ B$ does not impose any restrictions on A and B, the constraint is redundant. Usually, the number of primitive constraints is finite and they are pairwise disjoint. We will assume this in the following.

Definition 17. A network is *path consistent* if for pairs of nodes (i, j) and all paths $i - i_1 - i_2 \ldots i_n - j$ between them, the direct constraint c_{ij} is at least as tight as the indirect constraint along the path, i.e. the composition of constraints along the path denoted by $c_{ii_1} \otimes \ldots \otimes c_{i_n j}$. A constraint c_{ij} *is tighter* than a constraint d_{ij} iff c_{ij} implies d_{ij}.

It follows from the definition of path consistency that we can intersect the direct and indirect constraint to arrive at a tighter direct constraint. Let intersection be denoted by the operator $\oplus$. A graph is *complete* if there is a pair of arcs,

one in each direction, between every pair of nodes. If the graph underlying the network is complete it suffices to repeatedly consider paths of length 2 at most: For each triple of nodes (i, k, j) we repeatedly compute $c_{ij} := c_{ij} \oplus (c_{ik} \otimes c_{kj})$ until a fixpoint is reached. This is the basic path consistency algorithm.

Example 3. Given $I \leq K \ \wedge \ K \leq J \ \wedge \ I \geq J$ and taking the triple (i, k, j), $c_{ik} \otimes c_{kj}$ results in $I \leq J$ and the result of intersecting it with c_{ij} is $I = J$. From (j, i, k) we get $J = K$ (we can compute c_{ji} as the converse of c_{ij}). From (k, j, i) we get $K = I$. Another round of computation causes no more change, so the fixpoint is reached with $J = K \ \wedge \ K = I$.

Let the constraint c_{ij} be represented by the CHR constraint `c(I,J,C)` where `I` and `J` are the variables and `C` is a set of primitive constraints representing c_{ij}. The basic operation of path consistency, $c_{ij} := c_{ij} \oplus (c_{ik} \otimes c_{kj})$, can be implemented directly by the rule:

```
path_consistency @
c(I,K,C1), c(K,J,C2), c(I,J,C3) <=>
      composition(C1,C2,C12),intersection(C12,C3,C123),
      C123=\=C3 |
      c(I,K,C1), c(K,J,C2), c(I,J,C123).
```

The operations $\otimes$ and $\oplus$ are implemented by built-in constraints, `composition` and `intersection`. Composition of disjunctive constraints can be computed by pairwise composition of its primitive constraints. Intersection for disjunctive constraints can be implemented by set intersection. To achieve completeness, search must be employed. This is done by imposing primitive constraints chosen from the disjunctive constraints.

Termination. To prove termination we rely on the cardinality of the sets representing the disjunctive constraints and the properties of set intersection:

$$rank((A \wedge B)) = rank(A) + rank(B)$$
$$rank(c(I, K, C)) = cardinality(C)$$
$$rank(A) = 0 \text{ otherwise}$$

$$intersection(C1, C2, C3) \rightarrow rank(C3) \leq rank(C1) \wedge rank(C3) \leq rank(C2)$$
$$intersection(C1, C2, C3) \wedge C3 \neq C2 \rightarrow rank(C3) \neq rank(C2)$$

Since in the guard of the rule, `C123=\=C3` is checked to make sure the new constraint `C123` is different from the old one `C3`, the cardinality of `C123` must be strictly less than that of `C3`. Hence the body is ranked strictly smaller than the head of the rule. Goals are bounded, when C is a known, finite set of primitive constraints. Any solver derived from this generic path consistency solver will terminate, too.

4.5 Interval Constraints, Arc Consistency

The following rules implement an arc consistency algorithm for interval constraints [BeOl92]. Arc consistency can be seen as special case of path consistency, where all but one constraint is unary instead of binary. The interval constraint `X in A:B` means that `X` is an integer between the given bounds `A` and `B`.

```
% Intervals
inconsistent @ X in A:B <=> A>B | false.
intersection @ X in A:B, X in C:D <=> A=<B| X in max(A,C):min(B,D).

% (In)equalities
le @ X le Y, X in A:B, Y in C:D <=> A=<B,B>D |
     X le Y, X in A:D, Y in C:D.
le @ X le Y, X in A:B, Y in C:D <=> C=<D,C<A |
     X le Y, X in A:B, Y in A:D.

eq @ X eq Y, X in A:B, Y in C:D <=> A=<B,C=<D,A=\=C |
     X eq Y, X in max(A,C):B, Y in max(C,A):D.
eq @ X eq Y, X in A:B, Y in C:D <=> A=<B,C=<D,B=\=D |
     X eq Y, X in A:min(B,D), Y in C:min(D,B).

% Addition X+Y=Z
add @ add(X,Y,Z), X in A:B, Y in C:D, Z in E:F <=>
    A=<B,C=<D, not((A>=E-D,B=<F-C,C>=E-B,D=<F-A,E>=A+C,F=<B+D)) |
      add(X,Y,Z),
      X in max(A,E-D):min(B,F-C),
      Y in max(C,E-B):min(D,F-A),
      Z in max(E,A+C):min(F,B+D).
```

To achieve completeness, search must be employed. This is done by splitting intervals in two halves or by trying the boundary values of an interval.

Termination. We order constraints by the size of their intervals:

$rank((C \wedge D)) = rank(C) + rank(D)$
$rank(X \ in \ A : B) = B - A + 1$ if $B \geq A$
$rank(C) = 0$ otherwise

We will use the inequalities in the guards of the rules directly as order constraints. With their help we can prove that in each rule, at least one interval in the body is strictly smaller than the corresponding interval in the head, while the other intervals remain unchanged or will be removed.

The constraints `A=<B` and `C=<D` in the guard of a rule ensure that the rank of the head of the rule cannot be 0. (In implementations that apply rules in textual order, these guard constraints can be dropped.) The ranking condition for the first rule `inconsistent` also holds, even though its head rank is 0, since its order constraint is inconsistent: $(A > B \wedge \mathit{false}) \rightarrow 0 > 0$.

Since the interval bounds are initially known, goals are bounded. Note that the intervals of integers are closed under the interval computations used, since they involve only the arithmetic operations `max`, `min` and `+`, `-`. Termination for intervals of rational numbers can be shown by observing that any problem on rational numbers can be transformed into an equivalent one on integers by multiplying all numbers the problem with their greatest common divisor. For floating point numbers, rounding errors get in the way.

5 Conclusions

We have shown in this paper that for many known constraint solvers implemented in CHR it is possible to prove termination by adapting well-founded orders (linear polynomial interpretations) and interargument relations (order constraints) as known from related work in term rewriting systems and logic programming. One adaption was to extend termination order from atomic formulas to conjunctions of constraints.

To the best of our knowledge, this is the first report on proving termination of concurrent constraint logic programs and of constraint solver implementations. Our results indicate that in practice, proving termination for concurrent constraint logic programs may not be harder than for other classes of logic programming languages, contrary to what has been feared in the literature.

Our results so far are somewhat unsatisfactory, because we give two analogous Theorems which should be abstracted into single Theorem accomodating both kind of termination orders that we considered (polynomial interpreations and multi-sets).

The solvers we have considered are characterized by the fact that recursion is direct and typically modifies one argument position of a constraint, and the term in that position is sufficiently known in reasonable queries, i.e. those queries are bounded.

Although we have dealt with termination in the presence of propagation rules in the solver for terminological reasoning, we still have to formalize termination involving propagation rules. In particular, there is a class of solvers that we currently cannot prove to terminate with the approach presented in this paper. These solvers are implementations of path and arc consistency algorithms on incomplete constraint networks. They basically consist of the two following prototypical rules:

```
c(I,K,C1), c(K,J,C2) ==> composition(C1,C2,C3), c(I,J,C3).
c(I,J,C1), c(I,J,C2) <=> intersection(C1,C2,C3), c(I,J,C3).
```

These solvers have recursion on the same constraint through both simplification and propagation rules. This means that a constraint can be first added and then be removed during the computation. To prove termination, one will have to take into account that simplification is applied sufficiently often before propagation and the fact that propagation rules are never applied a second time to the same constraints.

Future work will also consider automation of the termination proofs presented here. Another interesting line of future work is to strengthen the antecedent of a ranking condition by introducing type constraints (since ill-typed goals either delay or fail).

Acknowledgements. The author would like to thank the anonymous reviewers for their valuable suggestions.

References

[Abd97] S. Abdennadher, Operational Semantics and Confluence of Constraint Propagation Rules, 3rd Intl Conf on Principles and Practice of Constraint Programming (CP'97), Linz, Austria, Springer LNCS 1330, pp 252-265, October/November 1997.

[AbFr99] S. Abdennadher and T. Frühwirth, Operational Equivalence of CHR Programs and Constraints, 5th Intl Conf on Principles and Practice of Constraint Programming (CP'99), Alexandra, Virginia, USA, Springer LNCS, 1999.

[AFM99] S. Abdennadher, T. Frühwirth and H. Meuss, Confluence and Semantics of Constraint Simplification Rules, Journal Constraints Vol. 4(2), Kluwer Academic Publishers, May 1999.

[ApPe90] K.R. Apt and D. Pedreschi, Studies in Pure Prolog: Termination, ESPRIT Computational Logic Symposium, Springer 1990, pp 150-176.

[BaHa91] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. 12^{th} International Joint Conference on Artificial Intelligence, 1991.

[BaNi98] F. Baader and T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[BeOl92] F. Benhamou and W.J. Older, Bell Northern Research, Applying interval arithmetic to Integer and Boolean constraints, Technical Report, June 1992.

[Bez93] M. Bezem, Strong Termination of Logic Programs, Journal of Logic Programming Vol. 15(1,2), pp. 79-98, 1993.

[CDJK99] H. Comon, M. Dincbas, J.-P. Jouannaud and C. Kirchner, A Methodological View of Constraint Solving, Constraints Journal Vol. 4(4), pp. 337-361, Kluwer Academic Publishers, December 1999.

[CMM95] L. Colussi, E. Marchiori and M. Marchiori, On Termination of Constraint Logic Programs, 1st Intl Conf on Principles and Practice of Constraint Programming (PPCP'95), Cassis, France, Springer LNCS 976, 1995.

[Der87] N. Dershowitz, Termination of Rewriting, Journal of Symbolic Computation Vol. 3(1+2), pp. 69-116, 1987.

[dSD94] D. de Schreye and St. Decorte, Termination of Logic Programs: The Never-Ending Story, Journal of Logic Programming Vol. 19,20, pp. 199-260, Elsevier, New York, USA, 1994.

[FrAb97] T. Frühwirth and S. Abdennadher, Constraint-Programmierung (in German), Textbook, Springer Verlag, Heidelberg, Germany, September 1997.

[FrBr00] T. Frühwirth and P. Brisset, Optimal Placement of Base Stations in Wireless Indoor Communication Networks, Special Issue of the IEEE Intelligent Systems Magazine on Practical Applications of Constraint Technology, (M. Wallace and G. Freuder, Eds.), IEEE Press, Vol. 15(1), pp. 49-53, January/February 2000.

[FrHa95] T. Frühwirth and P. Hanschke, Terminological Reasoning with Constraint Handling Rules, in Principles and Practice of Constraint Programming, (P. van Hentenryck and V.J. Saraswat, Eds.), MIT Press, Cambridge, Mass., USA, 1995.

[Fru98] T. Frühwirth, Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming (P. J. Stuckey and K. Marriot, Eds.), Journal of Logic Programming Vol. 37(1-3), pp. 95-138, Oct-Dec 1998.

[HoFr00] C. Holzbaur and T. Frühwirth, A Prolog Constraint Handling Rules Compiler and Runtime System, Special Issue of the Journal of Applied Artificial Intelligence on Constraint Handling Rules (C. Holzbaur and T. Fr"uhwirth, Eds.), Taylor & Francis, to appear 2000.

[Imb95] J.-L. J. Imbert, Linear Constraint Solving in CLP-Languages, in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), LNCS 910, March 1995.

[JaMa94] J. Jaffar and M. J. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming Vol. 19,20, pp. 503-581, 1994.

[KKS97] M.R.K. Krishna Rao, D. Kapur and R. K. Shyamasundar, Proving Termination of GHC Programs, New Generation Computing, 1997.

[MaFr85] A. K. Mackworth and E. C. Freuder, The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems, Journal of Artificial Intelligence Vol. 25, pp. 65-74, 1985.

[MaSt98] K. Marriott and P. J. Stuckey, Programming with Constraints, MIT Press, USA, March 1998.

[MaTe95] E. Marchiori and F. Teusink, Proving Termination of Logic Programs with Delay Declarations, ILPS 95, 1995.

[Mah87] M. J. Maher, Logic Semantics for a Class of Committed-Choice Programs, 4th Intl Conf on Logic Programming, Melbourne, Australia, pp 858-876, MIT Press, Cambridge, Mass., USA, 1987.

[Mes96] F. Mesnard, Inferring Left-terminating Classes of Queries for Constraint Logic Programs, Joint Intl Conf and Symposium on Logic Programming (JICSLP'96), (M. Maher, Ed.), pp 7-21, Bonn, Germany, MIT Press, September 1996.

[Nai92] L. Naish, Coroutining and the Construction of Terminating Logic Programs, Technical Report 92/5, Dept of Computer Science, University of Melbourne, Australia, 1992.

[Plu92] L. Plümer, Automatic Verification of GHC-Programs: Termination, Fifth Generation Computer Systems, (FGCS'92), Tokyo, Japan, June 1992.

[Rug97] S. Ruggieri, Termination of Constraint Logic Programs, ICALP 1997, Springer LNCS 1256, pp. 838-848, 1997.

[vHSD92] P. van Hentenryck, H. Simonis and M. Dincbas, Constraint Satisfaction Using Constraint Logic Programming, Artificial Intelligence, 58(1-3):113–159, December 1992.