Predicting Derivation Lengths in Rule-based Constraint 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 automatically predict the maximal number of rule applications, i.e. worst-case derivation lengths of computations, in rule-based constraint solver programs written in the CHR language. The derivation lengths are derived from rankings used in termination proofs for the respective programs. We are especially interested in rankings that give us a good upper bound, we call such rankings* tight. *We apply our method to constraint solvers ranging from Boolean and arithmetic to terminological and path-consistent constraints. In most cases, the worst-case derivation length is linear in the syntactic size of the constraint problem.*

RÉSUMÉ. *Nous prévoyons automatiquement le nombre maximal d'application de règles (c.-à.d. longueur de dérivation dans les pires des cas) dans des solveur de contraintes basés sur des règles écrits dans le langage CHR. Les longueurs de dérivation sont dérivées à partir des rangs utilisés dans des preuves de terminaison pour ces programmes. Nous sommes particulièrement intéressés par les rangs qui nous donnent une bonne limite supérieure. Nous appliquons notre méthode sur des differents solveurs de contraintes, tels que contraintes booléennes, équations linéaires et contraintes terminologiques. Dans la plupart des cas, la longueur de dérivation (dans les pires des cas) est linéaire dans la taille syntactique du problème.*

KEYWORDS: *Analysis, Termination, Constraint Solving, Constraint Handling Rules*

MOTS-CLÉS : *Analyse, Termination, Resolution de Contraintes, Constraint Handling Rules*

1. Introduction

CHR (Constraint Handling Rules) [Fru98, AFM99] 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 implement dozens of constraint solvers, including new domains such as terminological, spatial and temporal reasoning [Fru98].

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] and more recently also of constraint logic programs [CMM95, Mes96, Rug97]. The basic idea is to prove that in each rule, the head atom (left-hand side) is strictly larger than every atom occurring in the body of the rule (right-hand side).

In termination proofs for logic programs, often well-founded orders are adopted from term rewriting systems (TRS). A commonly used order is called *polynomial interpretation* [Der87, BaNi98]. The idea is to map terms and atoms to natural numbers using a polynomial function. Instances of this mapping are also called measure function, norm, ranking or level mapping in the literature.

A ranking proving termination maps head and body of each rule in a CHR program to natural numbers, such that the rank of the head is strictly larger than the rank of the body. A given constraint satisfaction problem is posed as a query to the CHR solver. Intuitively then, the rank of a query yields an upper bound on the number of rule applications (derivation steps), i.e. derivation lengths.

Example 1.1 Consider the following constraint `even` that ensures that a positive natural number in successor notation is even:

```
even(s(N)) <=> N=s(M),even(M).
```

The built-in constraint `=` stands for syntactical equality: `N=s(M)` ensures that `N` is the successor of some number `M`. The rule says that if the argument of `even` is the successor of some number `N`, then the predecessor of this number `M` must be even in order to ensure that the initial number `s(N)` is even. If a given problem constraint matches the head of a rule, it is replaced by the body of the rule. If no rule matches a constraint, the constraint delays.

The query `even(N)` delays. The query `even(0)` delays as well. To the query `even(s(N))` the rule is applicable, the answer is `N=s(M), even(M)`. The query `even(s(0))` will fail after application of the rule, since `0=s(M)` is inconsistent.

An obvious ranking is

$$rank(s(N)) = 1 + rank(N)$$
$$rank(0) = 0$$

The ranking gives us an upper bound on the derivation length, since with each rule application, we decrease the rank of the argument of `even` by 2.

The example also shows why we are interested in worst-case and not best-case derivation lengths: In constraint programming, there are queries (like `even(N)`) that delay, i.e. have derivation length zero, independent from their rank.

Related Work. To the best of our knowledge, there is no work in logic programming concerned with predicting derivation lengths for concrete programs. Somewhat related is [PiWi99], where an instance of quantative observables is used to prove termination of probabilistic CCP programs based on finite average derivation lengths. In the context of transforming CCP programs, where derivation length corresponds to the number of procedure expansions (unfolding steps), this measure is used to compare the efficiency of transformed programs in [BEP99]. In TRS, there is a line of theoretical work on predicting derivation lengths from termination orders. It is concerned with finding worst-case bounds for complex termination orders. A good starting point is the recent [Lep99] with pointers to the relevant publications.

What we present here are also practical results. From a suitable ranking for termination (introduced in [Fru99]), we are able to concisely predict the worst-case derivation lengths for a variety of actually implemented constraint solver programs written in CHR - ranging from Boolean and arithmetic to terminological and path-consistent constraints.

Overview of the Paper. We will first give syntax and semantics for CHR. In the next section, we summarize the basis for proving termination of CHR programs by introducing rankings and show how they can be used to derive tight upper bounds for worst-case derivation lengths. The main, fourth section reviews various CHR constraint solver programs and gives rankings for them. Based on the rankings, termination and derivation lengths are discussed. We conclude with a discussion of the results obtained.

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].

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. To keep the framework simple, we will regard all (auxiliary) predicates as built-in constraints.

The *concrete 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 2.1 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 *propagation 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 equality constraint `=` and the basic 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 *derivation steps (transitions, reductions)* one can proceed from one state to the next. A *derivation* is a sequence of derivation steps.

Definition 2.2 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 derivation step is possible anymore.

Definition 2.3 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 choosen from P. We use abstract syntax.

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 constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, 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, but not idempotent. Idempotency is avoided for both practical and theoretical reasons like efficiency of implementation and soundness of confluence results.

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 they match 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]. Another way of saying that H' matches H is that H' is an instance of H. 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, 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].

3. CHR Rankings

In this section, we summarize the basis for proving termination of CHR programs by introducing rankings and show how they can be used to derive tight upper bounds for worst-case derivation lengths.

3.1. *Termination by Rankings*

In [Fru99] we prove termination for CHR programs under any scheduling of rule applications (independent from the search and selection rule) using polynomial rankings.

Definition 3.1 A CHR program P is called *terminating for a class of queries*, if there are no infinite sequences of derivation steps using rules from P starting from a query in the class.

Roughly, a CHR program can be proven to terminate if we can prove that in each rule, the rank of the head is strictly larger than the rank of the body. To prove termination of CHR derivations, 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 appropriate rankings.

Definition 3.2 Let f be a function or predicate symbol of arity $n(n \geq 0)$ and let $t_i(1 \leq i \leq n)$ be terms. 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 for all terms and atoms t. An expression $rank(s) > rank(t)$ is called an *order constraint*.

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

Example 3.1 The *(syntactic) 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 $size(X) \geq 0$ 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.

A ranking for a CHR program will have to define the ranks of CHR and built-in constraints. We will define the rank of any built-in constraint to be 0, since we assume that they always terminate. A built-in constraint may imply order constraints between the ranks of its arguments (interargument relations), e.g. $s = t \rightarrow rank(s) = rank(t)$.

In extension of usual approaches, we have to define the rank of a conjunction of constraints, since CHR are multi-headed. The rank of a conjunction should reflect that conjunctions of CHR constraints are associative and commutative, but not idempotent. An obvious choice is $+$, i.e. the rank of a conjunction is the sum of the ranks of its conjuncts:

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

In [Fru99], we formalized 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 3.3 The *CHR ranking condition of a simplification rule* `H <=> G | B` is the formula

$$\forall\ (RC(G \wedge B) \rightarrow rank(H) > rank(B),$$

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

The intuition behind the definition of a ranking condition is that the built-in constraints in the rule will imply order constraints $RC(G \wedge B)$ that can help us to establish that $rank(H) > rank(B)$.

To ensure well-foundedness of the termination order, programs and queries either have to be well-moded (and well-typed) or queries sufficiently known. Since constraints do not go well with modes, the latter option is chosen.

Definition 3.4 A goal G is *bounded* if the rank of any instance of G is bounded from above by a constant k.

Intuitively, in bounded goals, variables only appear in positions which are ignored by the ranking. The rank of a ground (variable-free) term is always bounded.

The following Theorem tells us how to prove CHR program termination.

Theorem 3.1 [Fru99] Given a CHR program P without propagation rules. If the ranking condition holds for each rule in P, then P is terminating for all bounded goals.

Proof Sketch. Besides showing boundedness, we have to show that the rank of a state (goal) is strictly larger than the rank of its successor state (i.e. any state reachable by one derivation step):

Without loss of generality, consider a state $H' \wedge D$ and a rule (H <=> $G \mid B$) with ranking condition $RC(G \wedge B) \rightarrow rank(H) > rank(B)$. Applying the rule to the state $H' \wedge D$ will lead to the state $(H = H') \wedge G \wedge B \wedge D$. We show that $rank(H' \wedge D) > rank((H = H') \wedge G \wedge B \wedge D)$.

We know that $rank(G) = 0$, $rank(H = H') = 0$, since g and $H = H'$ are built-in constraints, and that $(H = H') \rightarrow rank(H) = rank(H')$.

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

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

3.2. *Derivation Lengths from Rankings*

The rank of a query gives us an upper bound on the number of rule applications (derivation steps), i.e. derivation lengths.

Theorem 3.2 Given a CHR program P without propagation rules. If the ranking condition holds for each rule in P, then the worst-case derivation length D_P for a bounded goal G in P is bounded by the rank of G. We write this as:

$$D_P \leq rank(G)$$

Proof. From the proof of Theorem 3.1 we know that given an derivation step $G \longmapsto G_1$ it holds that $rank(G) > rank(G_1)$. Since ranks are natural numbers, we may rewrite this as $rank(G) \geq rank(G_1) + 1$. By induction we can show that given a derivation of length n, $G \longmapsto^n G_n$, we have that $rank(G) \geq rank(G_n) + n$. Since ranks are non-negative, this implies the desired $rank(G) \geq n$.

We are interested in CHR rankings that get us as close as possible to the actual derivation lengths. This is the case if differences between the ranks of the heads and bodies of the rules in a program are bounded from above by a constant. We call such rankings *tight*.

Definition 3.5 Given a *CHR ranking of a simplification rule* `H <=> G | B`. The ranking is *exact for the rule* `H <=> G | B` iff $rank(H) = rank(B) + 1$.The ranking is *tight by* n *for the rule* `H <=> G | B` iff $rank(H) = rank(B) + n$, where the constant n is a natural number.The ranking is *tight by* n *for a CHR program* P iff the ranking is tight by n_i for all rules in P and n is the maximum of all n_i.

The definition of tightness is appropriate for worst-case analysis, while average-case analysis would have to take into account the distribution of the n_i.

4. Derivation Lengths of CHR Constraint Solvers

We now derive uppers bounds for the derivation lengths of actually implemented CHR constraint solvers ranging from Boolean and arithmetic to terminological and path-consistent constraints. We will use concrete syntax for the constraint solvers. For details on the solvers analyzed here see [Fru98] and the CHR web pages:
`www.pst.informatik.uni-muenchen.de/personen/fruehwir/chr/`

Based on the rankings, we will also prove termination and we relate the derivation length for a given query to the number of atomic constraints in the query, c, and to the number of different variables in the query, v (where possible).

4.1. *Boolean Algebra, Propositional Logic*

The domain of Boolean constraints [Me*93] 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. Syntactic equality `=` is a built-in constraint. In the constraint solver $Bool$, we define an `and` constraint using value propagation, a special case of arc consistency (see Section 4.3):

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

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 other logical connectives are defined analogously by single-headed simplification rules whose body contains only built-in constraints.

Termination. The above rules terminate, since the CHR constraints are not recursive. Any ranking that maps them to a positive number suffices. Also, any query is bounded.

Derivation Length. Since each CHR constraint is reduced to built-in constraints by a single rule application, the maximum derivation length is just the number of constraints in the query, c.

More formally, let the ranking be defined as

$rank(A) = 1$ if A is an atomic CHR constraint
$rank(A) = 0$ otherwise

For each rule in $Bool$ `H <=> G | B` we have that $rank(H) = 1$ and $rank(B) = 0$. Hence the ranking is exact for all rules. Consequently, the worst-case derivation length of a Boolean query is

$$D_{Bool} \leq c$$

It can be much smaller. For example, the query `and(U,V,W)` delays, its derivation length is zero. Another example is a query that contains the constraint `and(0,Y,1)`. If it is selected first, it will reduce to the inconsistent built-in constraint `1=0` in one derivation step. Because of the inconsistency, this is a final state of the derivation.

To establish a relationship between the derivation length and the number of different variables in a query, we assume that queries are of a certain *canonical form*, where the only arguments of the CHR constraints are variables and there are no multiple occurences of atomic CHR constraints.

Since the usual logical connectives are at most binary (like `and`), an atomic CHR constraint in $Bool$ can have at most three arguments. So given a query containing v different variables, we can generate at at most v^3 different constraints from it. Given k different logical connectives, we have that:

$$c \leq v^3 * k$$

Hence the derivation length can be cubic in the number of variables in the worst case:

$$D_{Bool} \leq v^3 * k$$

For example, given two variables `X` and `Y`, a largest query containing only `and` is

```
and(X,X,X), and(X,X,Y), and(X,Y,X), and(X,Y,Y),
and(Y,X,X), and(Y,X,Y), and(Y,Y,X), and(Y,Y,Y).
```

It will reduce to the constraint `X=Y` in 8 derivation steps.

Boolean Cardinality

The cardinality constraint combinator was introduced in the CLP language cc(FD) [vHSD92] for finite domains. In the solver $Card$ we adapted cardinality 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 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)`.

```
% 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).
```

In the program, all constraints except cardinality `#` are built-in. `all(T,L)` unifies all elements of the list `L` with `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. For example, `delete(1,BL,BL1)` will delay if it tries to unify a free variable in `BL` with `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), 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), the program is terminating.

Our ranking is based on the length of the list argument of `#`:

$$rank(\#(L, U, BL, N)) = 1 + length(BL)$$

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

The rank adds one to the length of the list in order to give a cardinality with the empty list a positive rank. For example, consider the query `#(0,0,[],0)`. Any of the three satisfaction rules can be applied to it and the derivation length will always be one.

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`:

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

Due to the ranking, a query 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.

Derivation Length. From the ranking we see that the derivation length of a single cardinality constraint is bounded by the length of the list arguments. For example, the query `#(1,1,[0,0,0,0,X],5)` needs five derivation steps to reduce to `X=1`. The first four steps remove the zeros from the list. The derivation length of a query is less or equal to the sum of the lengths of the lists occurring in the query. Hence it is linear in the syntactic size of the query in the worst case.

If the maximum length of the lists is bounded by $l - 1$, we have that:

$$D_{Card} \leq c * l$$

The ranking is exact for the two recursive reduction rules, because of the order constraint implied by `delete`. It is tight by l only for the three satisfaction rules, since a cardinality constraint with arbitrary rank may be reduced to built-in constraints with rank 0 in one derivation step. Hence the solver program *Card* is tight by l.

The number of variables in a query does not give an upper bound on the number of constraints for *Card*, since the lists of the cardinalities may contain an arbitrary number of (repeated) variables.

4.2. *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, MoHe86]. We use abstract syntax in the following definitions.

Definition 4.1 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.

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.

Usually, the number p of primitive constraints is finite and they are pairwise disjoint. We will assume so in the following.

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.

Definition 4.2 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 than the indirect constraint along the path, i.e. the composition of constraints $c_{ii_1} \otimes \ldots \otimes c_{i_n j}$ along the path.

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 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.

For example, 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 $I = J\ \wedge\ J = K\ \wedge\ K = I$.

Let the disjunctive binary constraint c_{ij} be represented in concrete syntax 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).
```

In this solver *Path*, the operations $\otimes$ and $\oplus$ are implemented by the 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. In the guard of the rule, the check `C123=\=C3` makes sure that the new constraint `C123` is different from the old one `C3`. Instances of a similar solver have been used for temporal reasoning [Fru94] and for spatial reasoning [EsTo98].

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

$rank(c(I, K, C)) = card(C)$
$rank(A) = 0$ otherwise.

$card(\{c_1, \ldots, c_n\}) = n$

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

Because of the properties of intersection and the guard check `C123=\=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. Queries are bounded, when C is a known, finite set of primitive constraints.

Derivation Length. Every rule application removes at least one primitive constraint and at most all of them from the set of primitive constraints `C3` by intersecting it with `C12`. If the maximum number of primitive constraints is p, the ranking is tight by at most p. The actual tightness depends on the intersection behavior of the set of primitive constraints.

For the derivation lengths we have that:

$$D_{Path} \leq c * p$$

i.e. the worst-case derivation length is linear in the syntactic size of the query.

Similar to the solver *Bool*, the number of variables gives an upper bound on the number of constraints in canonical queries where the first two arguments of each constraint are variables and where no duplicate constraints occur. Since the associated directed constraint graph must be complete, we have that:

$$c = v^2 * 2$$

Hence:

$$D_{Path} \leq v^2 * 2 * p$$

4.3. *Interval Constraints, Arc Consistency*

The following rules of the solver *Intv* implement an arc consistency algorithm for interval constraints (a special case of finite domain constraints) [vHSD92, vHDT92, Ben95]. The main idea of arc consistency is that it distinguishes a special class of unary constraints of the form $X \in D$, where D is a finite set of values.

Definition 4.3 A conjunction of unary constraints $X_1 \in D_1 \wedge \ldots \wedge X_n \in D_n$ is *arc consistent* with respect to a constraint $c(X_1, \ldots, X_n)$, if for all $i \in \{1, \ldots, n\}$ and for all possible valuations of X_i from its domain D_i the constraint $X_1 \in D_1 \wedge \ldots \wedge X_n \in D_n \wedge c(X_1, \ldots, X_n)$ is satisfiable.

In other words, in an arc consistent conjunction of constraints, every value of every domain takes part in a solution. A conjunction of constraints can be made arc consistent by deleting those values from the domain of the variables that do not participate in any solution of the constraints.

In our case, the domains are intervals of integers. The unary interval constraint `X in A:B` stands for $X \in \{n \mid n \in Z \wedge A \leq n \wedge n \leq B\}$. `in, le, eq` and `add` are CHR constraints, `<, =<, >, >=, <>` are built-in arithmetic constraints, and `min, max, +, -` are built-in arithmetic functions. Intervals of integers are closed under computations involving only these functions. The built-in prefix operator `not` negates its argument.

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

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

The rules affect the interval constraints only, the constraints `le, eq` and `add` remain unaffected. The rules `inconsistency` and `intersection` remove one interval constraint each. The built-in inequalities `A=<B` and `C=<D` used in the guards of the rules ensure that these rules apply only to consistent intervals. The remaining built-in inequalities in the guards ensure that in each rule, at least one interval gets strictly smaller.

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

$rank(X \ in \ A:B) = 2 + width(A:B)$
$rank(C) = 0$ otherwise.

$width(A : B) = B - A$ if $B \geq A$
$width(A : B) = -1$ if $B < A$

For the ranking, 2 is added to the interval width such that inconsistent and singleton intervals have positive ranks as well.

We use the inequalities in the guards of the rules directly as implied order constraints to show 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 unaffected. Since the interval bounds are initially known, queries are bounded.

Derivation Length. Let i be the the maximum rank of an interval constraint in a given query. The tightness of a rule can be computed by assuming that all interval constraints have maximum rank i except those whose intervals are computed in the body, they have minimum rank 1. The `inconsistency` rule is exact. For the reaming rules we have that $i > 1$. The `intersection` rule is tight by $2i - 1$, the rules for `eq` and `le` are tight by $i - 1$, the rule for `add` is tight by $3i - 3$. Hence the solver program *Intv* is exact for $i = 1$ and tight by $3i - 3$ for $i > 1$.

The derivation length is bounded by the sum of the interval sizes in a query:

$$D_{Intv} \leq c * i$$

Since the size of an interval depends on its bounds, the syntactic size of the query does not properly reflect the worst-case derivation length. In canonical queries, each variable is associated with exactly one interval constraint, and we have that:

$$v \leq c$$

but a tighter bound

$$D_{Intv} \leq v * i$$

since only interval constraints are affected by the solver.

4.4. *Terminological Reasoning, Description Logic*

Terminological formalisms (aka description logics) [PSR99] 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. In this paper, we use a natural language like syntax to help readers not familiar with the formalism.

Definition 4.4 *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 4.5 A *terminology (T-box)* consists of a finite set of acyclic *concept definitions*

c `isa` s,

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

The CHR constraint solver $Descr$ for description logics is similar to the one in [FrHa95], except that here we represent both the A-box and the T-box as constraints of the query. The solver simplifies and propagates assertions in the A-box by using the definitions in the T-box and by making information more explicit and looks for obvious contradictions such as `X : device` and `X : nota device`. This is handled by the rule:

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

The unfolding rules replace concept names by their definitions.

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

The conjunction rule generates two new, smaller assertions:

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

Disjunction is handled by lazy search, not directly by CHR. 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 using a propagation rule:

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

Since propagation rules are not covered in this paper, we ignore the rule. The final simplification rules push the complement operator `nota` down 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.
```

Termination. The only CHR constraints that are rewritten by the rules are membership assertions. Hence, it suffices to show that in each rule, the membership assertions in the body are strictly smaller than the ones in the head.

We rank constraints by the size of their concept terms:

$rank(c\ isa\ s) = 1$
$rank(I : s) = size(s)$
$rank(A) = 0$ otherwise

$size(nota\ s) = 2 * size(s)$
$size(some\ r\ is\ s) = 1 + size(s)$
$size(every\ r\ is\ s) = 1 + size(s)$
$size(c) = 1 + size(s)$ if `(c isa s)` exists
$size(f(t_1, \ldots, t_n)) = 1 + size(t_1) + \ldots + size(t_n)$ otherwise.

From the ranking we can see that queries 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 queries are bounded. For these reasons, the propagation rule can only generate a finite number of smaller and smaller membership assertions.

Derivation Length. The derivation length D_{Descr} is bounded by the sum of the sizes of the concept terms occuring in a query. Since the size of a concept depends on its definition, the syntactic size of the query does not properly reflect the worst-case derivation length.

Let the maximum size of a concept term be bounded by a constant k. The ranking is exact for all but three rules: the rule involving complement and concept definition, which is tight by 2, the rule handling contradiction (tight by at most $3k/2$) and the rule for double complement (tight by at most $3k/4$).

As long as lazy search for disjunction and the propagation rule for value restrictions is not involved, we have that

$$D_{Descr'} \leq c * k$$

Again, the propagation rule

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

needs closer consideration. To calculate the worst-case bound, consider a query with one constraint `I : every R is S` and c identical CHR constraints `(I,I) : R`.

By applying the propagation rule c times, we generate c additional new CHR constraints `I : S`.

Now let `S` be of the finite form `every R is every R is ...is S1`. Then from the c new CHR constraints `I : S` and the c given constraints `(I,I) : R` we can in turn generate $c * c$ new constraints by $c * c$ rule applications. This multiplication by c can be repeated at most $k - 1$ times, where k is the initial size of `every R is S`. The sum of this geometric series captures the exponential blow-up that can be caused by the propagation rule.

Consequently, the behavior of the propagation rule dominates the worst-case derivation lengths for $c > 1$:

$$D_{Descr} \leq c^k$$

The number of variables v is not directly related to D_{Descr}, since a variable can occur in several membership assertions and since exists-in restrictions generate new variables. and since one variable is enough to enable the worst-case behavior of the propagation rule.

5. Conclusions

We predicted the maximal number of rule applications, i.e. worst-case derivation lengths of computations, in CHR constraint solver programs. The derivation lengths are derived from tight rankings used in termination proofs.

Usually, the worst-case derivation length D is linear in the size of the constraint problem (query), the only exception is the solver for description logics *Descr*. For these solvers, D is bounded by $c * MaxSize$, where c is the number of atomic constraints in the query and *MaxSize* is the maximum size (rank) of an atomic constraint. Except for the solver *Descr* and the interval solver *Intv*, the *syntactic* size of a query properly reflects its worst-case derivation length.

Except for the solver *Descr* and the solver for Boolean cardinality *Card*, D is polynomial in the number of different variables, v, in *canonical* queries. More precisely, D is bounded by $v^n * MaxSize$, where n is the maximum arity of the relevant constraints in the solver.

The solver *Descr* impressingly shows the potential danger of propagation rules. Due to a single such rule, the solver goes from a linear to an exponential bound for D. In our current framework, propagation rules are handled in an ad-hoc way. Future work is concerned with extending our method for proving termination and predicating derivation lengths to propagation rules.

One may ask if the derivation length properly reflects the time complexity of a CHR program. This is not necessarily the case, as the solvers for path consistency and interval arc consistency, *Path* and *Intv*, show: According to [MaFr85, MoHe86], the worst-case time complexity of path consistency is $O(v^3 * p^3)$, we have $v^2 * 2 * p$ only.

Arc consistency is $O(v^2 * i^2)$, we have $v * i$. Typically, the order of the derivation length is less than the order of time complexity. The main reason is that our measure does not take into account the effort of finding the appropriate combination of constraints in the query that match the multi-head of a rule. Future work should investigate this issue together with providing empirical results.

6. References

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

[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.

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

[Ben95] F. Benhamou, Interval Constraint Logic Programming, in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, March 1995.

[BEP99] M. Bertolino, S. Etalle and C. Palamidessi, The Replacement Operation for CCP Programs, 9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99), Venice, Italy, September 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.

[CoDi96] P. Codognet and D. Diaz, Compiling constraints in `clp(fd)`, Journal of Logic Programming, 27(3), 1996.

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

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

[EsTo98] M.T. Escrig and F. Toledo, Qualitative Spatial Reasoning: Theory and Practice, IOS Press, 1998.

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

[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.

[Fru94] T. Frühwirth, Temporal Reasoning with Constraint Handling Rules, Technical Report ECRC-94-05, ECRC Munich, Germany, February 1994.

[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):95-138 Oct-Dec 98.

[Fru99] T. Frühwirth, Termination of CHR Constraint Solvers, ERCIM Working Group on Constraints / CompulogNet Area on Constraint Programming Workshop, Paphos, Cyprus, October 1999.

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

[Lep99] I. Lepper, Simple Termination is Complex, Journal preprint, Westfälische Wilhelms-Universität Münster, Germany, June 1999.

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

[Me*93] S. Menju et al., A Study on Boolean Constraint Solvers, Constraint Logic Programming: Selected Research, (F. Benhamou and A. Colmerauer, Eds.), MIT Press, Cambridge, Mass., USA, 1993.

[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.

[MoHe86] R. Mohr and T.C. Henderson, Arc and Path Consistency Revisited, Artificial Intelligence 28: 225-233, 1986.

[PiWi99] A. Di Pierro and H. Wiklicky, Quantitative Observables and Averages in Probabilistic Constraint Programming, ERCIM Working Group on Constraints / CompulogNet Area on Constraint Programming Workshop, Paphos, Cyprus, October 1999.

[PSR99] P. Patel-Schneider and M-C. Rousset, Eds., Special Issue on Description Logics, Journal of Logic and Computation, Volume 9, Number 3, Oxford University Press, June 1999.

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

[vHDT92] P. Van Henteryck, Y. Deville and C.-M. Teng, A generic arc-consistency algorithm and its specializations, Artificial Intelligence, 57:291-321, 1992.

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

[vHSD95] P. van Hentenryck, V. A. Saraswat, and Y. Deville, Constraint Processing in cc(FD), Chapter in Constraint Programming: Basics and Trends, (A. Podelski, Ed.), Springer LNCS 910, 1995.