Complexity of the CHR Rational Tree Equation Solver

Marc Meister and Thom Frühwirth

Fakultät für Informatik, Universität Ulm, Germany
{Marc.Meister, Thom.Fruehwirth}@uni-ulm.de

Abstract. Constraint Handling Rules (CHR) is a concurrent, committed-choice, rule-based language. One of the first CHR programs is the classic constraint solver for syntactic equality of rational trees that performs unification [7, 4, 14]. The worst-case time (and space) complexity of this short and elegant solver so far was an open problem [8] and assumed to be *polynomial*. In this paper we show that under the standard operational semantics of CHR there exist particular computations with n occurrences of variables and function symbols that produce $O(2^n)$ constraints, thus leading to *exponential* time and space complexity. We also show that the standard implementation of the solver in CHR libraries for Prolog may not terminate due to the Prolog built-in order used in comparing terms. Complexity can be improved to be *quadratic for any term order* under both standard and refined CHR semantics without changing the equation solver, when equations are transformed into flat normal form.

1 Introduction

Unification Algorithms. Unification is concerned with making first order logic terms syntactically equivalent by substituting terms for variables. For example, the terms $h(a, f(Y))$ and $h(Y, f(a))$ can be made identical by substituting the variable Y by a. In 1930, Herbrand [9] gave an informal description of a unification algorithm. Robinson [13] rediscovered a similar algorithm when he introduced the resolution procedure for first-order logic in 1965. Resolution and unification form the computational basis of logic programming languages such as Prolog. Since the late 70s, there are quasi-linear time algorithms for unification. For finite trees (Herbrand terms), see [11] and [12]. For rational trees, see [10]. These algorithms can be considered as extensions of the union-find algorithm [17] from constants to trees.

Syntactic Equations. In constraint programming, unification of terms is understood as solving equations, e.g., the equation $h(a, f(Y))$ `eq` $h(Y, f(a))$ will reduce to the solved normal form Y `eq` a. Syntactic equality is an essential ingredient of constraint logic programming, since terms are the universal data structure and equalities can be used to build, access, and take apart terms. In early Prolog implementations, the *occur-check* was omitted from syntactic equality for efficiency reasons. The result was that equation solving could go into an infinite loop (e.g. for X eq $f(X)$). In Prolog II, an algorithm for properly handling the resulting infinite terms was introduced [3]. This class of infinite terms is called rational trees and is introduced in Section 2.

CHR Rational Tree Solver. Constraint Handling Rules CHR [4, 8, 14] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulas) into simpler ones until they are solved. Like other algorithms for rational tree unification, the CHR rational tree solver (c.f. Section 3, Fig. 1) relies on a size-based order on terms to ensure termination. However, a formal termination proof for the solver is not available so far[1]. Standard proof methods and counter-examples for unification algorithms do not apply, since the CHR solver does not rely on solved variables and their substitutions. Even worse, the standard implementation of the solver uses the Prolog term order. We show that the solver does not terminate with that particular order.

Exponential Complexity. In Section 4, we investigate termination and worst-case time and space complexity of the solver when using a certain *measure order*. It is based on a measure that maps terms and constraints to natural numbers. To the best of our knowledge, this yields the first termination proof for an unification algorithm where a scalar suffices (usually lexicographic or multi-set orderings are used). Our main result is that there are computations under the standard CHR operational semantics for a problem with size n that require $O(2^n)$ rule applications in the worst-case. This exponential complexity is shown to be tight. To this end, we give a witness query with size $O(n^2)$ that produces more than 2^n constraints. Therefore, worst-case space and (hence) time complexity of the classic CHR constraint solver for unification is *exponential*. However, it is still an open problem, if the result carries over to actual implementations of the solver that usually rely on the refined CHR semantics.

Quadratic Complexity. But we also have good news in Section 5: The very same solver runs in quadratic time and space, as we prove, by requiring that the equations to be solved are in flat normal form. Such equations do not contain terms with nested applications of function symbols. For example, the equation $h(a, f(Y))$ eq $h(Y, f(a))$ can be flattened into $h(A, Z)$ eq $h(Y, X) \wedge A$ eq $a \wedge Z$ eq $f(Y) \wedge X$ eq $f(A)$. Since any set of syntactic equations can be transformed into flat normal form in linear time and space, this requirement is no real restriction. We also show that the results are rather independent of the term order used in the solver.

2 Rational Trees

A *rational tree* has a finite representation as a directed (possibly cyclic) graph by merging all nodes with common subtrees.

Definition 1. *A* rational tree *(or RT) is a (possibly infinite) finitely branching tree which has a finite number of subtrees.*

[1] This may be an example were elegance does not pay off: The solver consists of just four rules, and so is more concise than most formal specifications of unification.

A rational tree can also be represented as a binary equality constraint. For example, the infinite tree $f(f(f(\ldots)))$ only contains itself and can be represented by the equation $X \ \texttt{eq} \ f(X)$. (Variables are written in upper-case and function symbols in lower-case letters as common in logic programming.)

A conjunction of atomic constraints is *solved* (or in *solved normal form*) if it is either *false* or if it is of the form $\bigwedge_{i=1}^{n} X_i \ \texttt{eq} \ T_i$ with pairwise distinct variables $X_1, \ldots, X_n$ and arbitrary terms $T_1, \ldots, T_n$ for $n \in \mathbf{N}$. We require X_i to be different to T_j for $1 \leq i \leq j \leq n$, i.e., if a variable occurs on the l.h.s. of an equation, it does neither occur as the l.h.s. nor r.h.s. of any subsequent equation. The empty conjunction ($n = 0$) is identified with *true*. For example, the equations $f(X, b) \ \texttt{eq} \ f(a, Y)$, $X \ \texttt{eq} \ t \wedge X \ \texttt{eq} \ s$, and $X \ \texttt{eq} \ Y \wedge Y \ \texttt{eq} \ X$ are all *not in solved form*, while $X \ \texttt{eq} \ Z \wedge Y \ \texttt{eq} \ Z \wedge Z \ \texttt{eq} \ t$ is *in solved form*. The solved form is not unique, e.g., $X \ \texttt{eq} \ Y$ and $Y \ \texttt{eq} \ X$ are logically equivalent but syntactically different solved forms, as are $X \ \texttt{eq} \ f(X)$ and $X \ \texttt{eq} \ f(f(X))$.

3 Rational Tree Equation Solver

The CHR program in Fig. 1 solves rational tree equations [7, 4, 8]. This solver dates back to late 1993 and was revised in 1998 [14]. The underlying algorithm is similar to the one in [3], but unlike this and most other unification algorithms it uses variable elimination (substitution) only in a very limited way, if it cannot be avoided. As a consequence, the algorithm has to rely on an order on variables. However, this simplification of the algorithm makes termination and complexity analysis considerably harder.

The auxiliary built-ins of the solver allows one to be independent of the representation of terms in the implementation: Besides *true* and *false*, we have var(T) iff T is a variable and fun(T) iff T is a function term (i.e. not a variable). A generic total order is implemented by $\prec$ and $\preceq$ and explained below in Subsection 3.2. The built-in `same_functor(T1,T2)` tests if `T1` and `T2` have the same function symbol and the same arity. It leads to *false* if not. The auxiliary CHR constraint `same_args(T1,T2)` pairwise equates the arguments of the two terms.

The rule `reflexivity` removes trivial equations between identical variables. The rule `orientation` reverses the arguments of an equation so that the (smaller) variable comes first. The order check in the guard makes sure that it is applicable only once to a given equation. The rule `decomposition` applies to function terms. When there is a clash, `same_functor` will lead to *false*. Otherwise, the initial equation between two function terms will be replaced by equations between the corresponding arguments of the terms. The rule `confrontation` replaces the variable in the second equation by the value of that variable according to the first equation. It performs a limited amount of variable elimination (substitution) by only considering the l.h.s.' of equations. This rule duplicates the term `T1`. For termination it is ensured by the guard that `T1` is not larger than `T2`.

Due to the `confrontation` rule, the complexity of the solver is worse than linear. The intricate interaction between the `decomposition` rule and the `con-`

`frontation` rule in the case of infinite terms (cyclic terms) makes it hard to determine the worst-case time complexity of the solver.

```
reflexivity   @ X eq X           <=> var(X) | true.
orientation   @ T eq X           <=> var(X), X≺T | X eq T.
decomposition @ T1 eq T2         <=> fun(T1), fun(T2) |
                                     same_functor(T1,T2), same_args(T1,T2).
confrontation @ X eq T1, X eq T2 <=> var(X), X≺T1, T1⪯T2 |
                                     X eq T1, T1 eq T2.
```

Fig. 1. Rational tree equation solver (RT solver)

The solver is satisfaction-complete, i.e. detects unsatisfiability: The conditions for the solved normal form can be restated as $X_i \prec X_{i+1}$ and $X_i \prec T_{i+1}$ (for $1 \leq i < n$) since any strict total order is transitive and asymmetric. Actually, the solver computes the solved form, as can be shown by contradiction: As long as a conjunction of constraints is not in solved form, at least one rule is applicable. If it is in solved form, no rule is applicable.

Example 1. Here is a simple example involving infinite rational trees that shows that one of the equations is redundant.

$$
\begin{array}{ll}
 & \underline{\texttt{X eq f(X)}},\ \underline{\texttt{X eq f(f(X))}} \\
\mapsto_{\texttt{confrontation}} & \texttt{X eq f(X)},\ \underline{\texttt{f(X) eq f(f(X))}} \\
\mapsto_{\texttt{decomposition}}\mapsto^* & \underline{\texttt{X eq f(X)}},\ \underline{\texttt{X eq f(X)}} \\
\mapsto_{\texttt{confrontation}} & \texttt{X eq f(X)},\ \underline{\texttt{f(X) eq f(X)}} \\
\mapsto_{\texttt{decomposition}}\mapsto^* & \texttt{X eq f(X)},\ \underline{\texttt{X eq X}} \\
\mapsto_{\texttt{reflexivity}} & \texttt{X eq f(X)}
\end{array}
$$

3.1 Term Size and Problem Size

We first define term and problem size and then the generic order $\prec$ based on term size that is used in the RT solver to compare terms.

Definition 2. *The* term size $\#T$ *of a term* T *is the number of occurrences of variables and function symbols. For two function terms* S *and* T*, we define the* term-size order $S \prec_s T$ *iff* $\#S < \#T$*. The* problem size $\#C$ *of a conjunction* $C = \bigwedge_{i=1}^{n} S_i \ \texttt{eq} \ T_i$ *of equations with* $n \in \mathbf{N}$ *is defined as* $\#\left(\bigwedge_{i=1}^{n} S_i \ \texttt{eq} \ T_i\right) := \sum_{i=1}^{n} \#S_i + \#T_i$.

For example, the problem size of (the empty conjunction) *true* is 0, the size of $X \ \texttt{eq} \ f(a)$ is 3, and $X \ \texttt{eq} \ f(b) \wedge f(b) \ \texttt{eq} \ h(a)$ has size 7.

3.2 Generic Term Order and Termination

Ordering terms according to the number of occurrences of symbols is common in the rational tree literature.

Definition 3. *Any instance of the* generic strict total term order $\prec$ *must have the following three properties [8]:*

1. *For different variables X and Y, either $X \prec Y$ or $Y \prec X$.*
2. *Any variable is strictly smaller than any function term.*
3. *Function terms of smaller term size are also smaller in the order (term-size property).*

A function term *is a term that is not a variable. Two terms T_1 and T_2 are* equivalent w.r.t. $\prec$ *if neither $T_1 \prec T_2$ nor $T_2 \prec T_1$. Clearly, terms of same size may be syntactically different terms.*

Termination of the RT solver crucially relies on the generic order $\prec$. The rules `reflexivity` and `orientation` are applicable at most once to an equation. Application of `decomposition` produces equations between the arguments of the functions of the initial equations. Thus, the new equations have arguments of smaller size. Application of `confrontation` replaces one occurrence of X by T_1. The guard ensures that $\texttt{X} \prec T_1 \preceq T_2$. Therefore, as long as T_1 is a variable, it gets closer from below to T_2 but can never exceed it. If T_1 is a function term, then so must be T_2, and then only `decomposition` is applicable to the new equation T_1 `eq` T_2 produced by `confrontation`. The resulting equations, including the unchanged X `eq` T_1, will only contain terms that are strictly smaller than T_2. Since there is only a finite number of variables and sub-terms in a given problem and since the generic term order is thus well-founded, the solver terminates[2].

3.3 Non-Termination with Standard Prolog Order

The standard implementation of the RT solver [8, 14] uses the built-in Prolog order `@<`. Variables are identified by the built-in `var/1` and function terms by the built-in `nonvar/1`. The Prolog order `@<` compares arguments of function terms lexicographically from left to right, e.g., $f(Y, f(a, X))$ `@<` $f(a, X)$ but $f(a, X) \prec_s f(Y, f(a, X))$. The order `@<` therefore does not respect the term-size property, it is *not* an instance of the generic term order. As we show in the following example, this can cause non-termination for infinite rational trees:

Example 2. The query `X eq f(Y,f(a,X)), X eq f(a,X)` does not terminate.

$$
\begin{array}{ll}
 & \underline{\texttt{X eq f(Y,f(a,X))}},\ \underline{\texttt{X eq f(a,X)}} \\
\mapsto_{\texttt{confrontation}} & \underline{\texttt{X eq f(Y,f(a,X))}},\ \underline{\texttt{f(Y,f(a,X)) eq f(a,X)}} \\
\mapsto_{\texttt{decomposition}}\mapsto^{*} & \texttt{X eq f(Y,f(a,X))},\ \texttt{Y eq a},\ \underline{\texttt{f(a,X) eq X}} \\
\mapsto_{\texttt{orientation}} & \texttt{X eq f(Y,f(a,X))},\ \texttt{Y eq a},\ \overline{\texttt{X eq f(a,X)}}
\end{array}
$$

Similarly, and containing only one binary function symbol, the computation for the query `X eq f(Y,f(f(X,Y),X)), X eq f(f(X,Y),X)` does not terminate. Note that in the next section we give another order that is also incompatible with the generic term order, but makes the RT solver provably terminate.

[2] A formal termination proof for the solver with generic term order is still missing.

4 Exponential Complexity

In this section we show that there exists a term order for the RT solver such that the worst-case time and space complexity of the solver can be exponential in the size of the problem. This term order, however, is not an instance of the generic term order commonly used in the solver.

We define a *problem measure*, which maps CHR constraints into natural numbers. It is based on a term measure that is exponential in the depth of the term. In the RT solver, we replace the generic order $\prec$ by the so-called *measure order* which is defined in terms of the measure.

We show for each rule that the problem measure of the head is always strictly greater than the problem measure of the body, provided the guard holds. In this way we not only formally prove termination, but also show that the problem measure gives us an upper bound on the number of rule applications (derivation length) [5]. Since the cost of a rule application can be made constant in the RT solver, the derivation length directly gives us the desired complexity result. A worst-case example then shows that the bound is actually tight.

4.1 Term Measure and Problem Measure

We give an inductive definition of our *term measure*[3]. As the RT solver does not introduce *new* variables, the number of different variables v of a problem is clearly bounded by its size. Hence, we can assume that all variables are elements of $\{X_1, \ldots, X_v\}$. The natural number v is called the *number of variables* of the problem.

Definition 4. *The* term measure $|T|$ *of a term T is defined as follows:*

$$|T| := \begin{cases} i & \textit{if } T = X_i \\ n + 2\sum_{i=1}^{n} |T_i| & \textit{if } T = f(T_1, \ldots, T_n) \end{cases}$$

Note that a constant (i.e., a null-ary function) has measure 0. Due to the factor 2 in the recursive definition, the measure of a term *can be exponential* in its size, consider, e.g., the term `f(f(f(a)))`.

We extend our term measure to equations and queries.

Definition 5. *For an equation S `eq` T with terms S and T, we define the* constraint measure[4]*.*

$$|S \texttt{ eq } T| := 1 + |S| + |T| + \begin{cases} |T| & \textit{if } \operatorname{var}(S) \wedge \operatorname{fun}(T) \\ |S| & \textit{if } \operatorname{var}(T) \wedge \operatorname{fun}(S) \\ 0 & \textit{otherwise} \end{cases}$$

[3] Term measures are also called *norms* in the literature on termination of constraint logic programs.

[4] Constraint measures are also called *level mappings* and *ranks* in the literature on termination of constraint logic programs.

The constraint measure summation consists of three components. The first component counts each equation as 1. The number of equations decreases when the rule `reflexivity` or `decomposition` of constants is applied. The second component $|S| + |T|$ accounts for the sizes of the arguments of the equations. It decreases when rule `decomposition` is applied. The third component adds to a variable in one argument the size of the other argument. It is introduced to handle the rule `confrontation`, where a variable is replaced by the term in its other argument. This reasoning will be made more formal in the next subsection when we compute the derivation length.

Definition 6. *For a conjunction $\bigwedge_{i=1}^{n} S_i \ \mathtt{eq}\ T_i$ of equations (which we call problem C), consisting of terms S_i and T_i for $1 \leq i \leq n$ and $n \in \mathbf{N}$, we define the* problem measure $|C| := \sum_{i=1}^{n} |S_i \ \mathtt{eq}\ T_i|$. *The problem measure is extended to any conjunction of constraints by ignoring any occurrence of the built-in constraints true and false, i.e., $|false| := |true| := 0$.*

Clearly, the *problem measure* is invariant to *reordering* and to *orientation* of equations. The problem measure decreases if one of its contributing constraint measures decreases. Thus, local replacements of equations, caused by a rule applications, can be treated independently.

4.2 Measure Order

Now we replace the generic order of the RT solver by a *measure order* which is defined via the term measure.

Definition 7. *The* measure order $\prec_m$ *induced by the term measure is defined by the three cases:*

1. *For two variables X and Y: $X \prec_m Y$ iff $|Y| < |X|$*
2. *For any variable X and any function term T: $X \prec_m T$*
3. *For two function terms S and T: $S \prec_m T$ iff $|S| < |T|$*

We emphasise that variables X_i and X_j are ordered by *decreasing term measure*, $X_i \prec_m X_j$ iff $|X_j| < |X_i|$ iff $j < i$, while function terms S and T are ordered by *increasing term measure*, $S \prec_m T$ iff $|S| < |T|$. The reverse ordering of variables will come handy when reasoning about the rule `confrontation`.

In the sequel, we assume the RT solver of Fig. 1 uses our measure order $\prec_m$ (c.f. Definition 7). Note that the measure order $\prec_m$ is *not* an instance of the generic term order $\prec$. For example, $f(f(f(a))) \prec_s f(a, a, a, a)$ in term-size order while $f(a, a, a, a) \prec_m f(f(f(a)))$ in measure order.

4.3 Number of Rule Applications

We will need the following inequality between the constraint measure and its arguments' term measures.

Lemma 1 *For any two terms S and T, the constraint measure is bounded by twice the sum of its term measures plus one:* $|S \ \mathtt{eq}\ T| < 2\big(1 + |S| + |T|\big)$.

Proof. Directly by Definition 5. □

The problem measure gives an upper bound on the derivation length.

Lemma 2 *Each application of one of the rules* ***`reflexivity`****,* ***`decomposition`****, and* ***`confrontation`*** *decreases the problem measure.*

Proof. We consider each rule in turn.

Application of `reflexivity`: Consider any variable X.

$$|X \ \mathtt{eq}\ X| = 1 + |X| + |X| \ > \ 0 = |\mathit{true}|$$

Application of `decomposition`: First, consider the case of two function terms with same function symbols and same arities.

$$\begin{aligned}
|f(S_1,\dots,S_n) \ \mathtt{eq}\ f(T_1,\dots,T_n)| &= 1 + |f(S_1,\dots,S_n)| + |f(T_1,\dots,T_n)| \\
&= 1 + \Big(n + 2\sum_{i=1}^{n}|S_i|\Big) + \Big(n + 2\sum_{i=1}^{n}|T_i|\Big) \\
&= 1 + \sum_{i=1}^{n} \underbrace{2(1 + |S_i| + |T_i|)}_{\underset{\text{Lemma 1}}{>} |S_i \ \mathtt{eq}\ T_i|} \\
&> \sum_{i=1}^{n} |S_i \ \mathtt{eq}\ T_i| = \Big|\bigwedge_{i=1}^{n} S_i \ \mathtt{eq}\ T_i\Big|
\end{aligned}$$

Note that decomposition of constants (i.e., null-ary function symbols) is also covered. Second, when the two function terms have different function symbols or different arities ($f \neq g$ or $m \neq n$) there is a *clash*. Then the RT solver immediately returns *false*.

$$\begin{aligned}
|f(S_1,\dots,S_m) \ \mathtt{eq}\ g(T_1,\dots,T_n)| &= 1 + |f(S_1,\dots,S_m)| + |f(T_1,\dots,T_n)| \\
&> 0 = |\mathit{false}|
\end{aligned}$$

Application of `confrontation`: We consider all three cases in turn (due to the restrictions by the guard there is no fourth case). For a variable X and two arbitrary terms T_1 and T_2, the guard requires $X \prec_m T_1$ and $T_1 \preceq_m T_2$.

- For two variables T_1 and T_2, we have $|X| > |T_1|$ (because $X \prec_m T_1$)

$$|\mathtt{X}\ \mathtt{eq}\ T_2| = 1 + |X| + |T_2| > 1 + |T_1| + |T_2| = |T_1 \ \mathtt{eq}\ T_2|$$

- For a variable T_1 and a function term T_2, we have $|X| > |T_1|$ (because $X \prec_m T_1$)

$$|\mathtt{X}\ \mathtt{eq}\ T_2| = 1 + |X| + 2|T_2| > 1 + |T_1| + 2|T_2| = |T_1 \ \mathtt{eq}\ T_2|$$

– Finally, for two function terms T_1 and T_2, we have $|T_1| \leq |T_2|$ (because $T_1 \preceq_m T_2$)

$$|\texttt{X eq } T_2| = 1+|X|+2|T_2| > 1+|T_2|+|T_2| \geq 1+|T_1|+|T_2| = |T_1 \texttt{ eq } T_2|$$

In all three cases, application of the corresponding rule decreases the problem measure. □

Lemma 3 *The number of rule applications (for all four rules) is bounded by twice the problem measure.*

Proof. By Lemma 2, the problem measure is an upper bound for the number of rule applications of `reflexivity`, `decomposition`, or `confrontation`. The problem measure is invariant to orientation of equations. As `orientation` can apply at most *once* to each available constraint, there are at most *twice* as many rule applications by all four rules than the problem measure. □

4.4 Tightness of the Problem Measure

To exhibit the worst-case, the query should decrease the problem measure as little as possible, i.e. the strict inequalities in the proof of Lemma 2 should be as tight as possible. We can see that we should use as few variables as possible; avoid a clash; make sure that after decomposition, the new equations are between a variable and a function term; confront terms with the same measure if possible.

Definition 8. *For the variable X and the binary function symbol f we define the following, mutually recursive terms for all natural numbers $n \in \mathbf{N}$.*

$$\mathcal{U}_n := \begin{cases} X & \text{if } n = 0 \\ f(\mathcal{L}_{n-1}, X) & \text{otherwise} \end{cases} \qquad \mathcal{L}_n := \begin{cases} X & \text{if } n = 0 \\ f(X, \mathcal{U}_{n-1}) & \text{otherwise} \end{cases}$$

For example, for $n = 4$ we have $\mathcal{U}_4 = f(f(X, f(f(X,X),X)),X)$ and $\mathcal{L}_4 = f(X, f(f(X, f(X,X)),X))$.

Lemma 4 (Properties of $\mathcal{U}_n$ and $\mathcal{L}_n$) *For $n \in \mathbf{N}$:*

1. $\#\mathcal{U}_n = \#\mathcal{L}_n$ *and* $|\mathcal{U}_n| = |\mathcal{L}_n|$.
2. *The term size is* linear*:* $\#\mathcal{U}_n = 2n + 1$.
3. *The term measure is* exponential*:* $|\mathcal{U}_n| \geq 2^n$.
4. $\mathcal{L}_n \preceq_m \mathcal{U}_n$ *and* $\mathcal{U}_n \preceq_m \mathcal{L}_n$

Proof. The easy inductions are omitted for lack of space.

Because the terms $\mathcal{L}_n$ and $\mathcal{U}_n$ are equivalent w.r.t. to measure order $\prec_m$ we can give a computation, that produces $\mathcal{L}_{n-1}$ `eq` $\mathcal{U}_{n-1}$ from $\mathcal{L}_n$ `eq` $\mathcal{U}_n$. In detail, we provide a query consisting of such equations which has exponential derivation length.

Lemma 5 (Exponential query) *For $n \in \mathbf{N}^+$ consider the following query $Q(n)$.*

$$\left(\bigwedge_{i=1}^{n} X \texttt{ eq } \mathcal{L}_i\right) \wedge X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n$$

Query $Q(n)$ has quadratic size $\#Q(n) = O(n^2)$. There exists a computation in standard semantics for $Q(n)$ which produces exponentially many equations: precisely, 2^{n+1} equations X `eq` X are produced.

We delay the proof of Lemma 5 and introduce a *sub-computation* $S(n)$ (for a given $n \in \mathbf{N}^+$). Sub-computation $S(n)$ can be applied to states that contain c copies[5] of the conjunction $X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n$ plus one additional *catalyst* copy of $X \texttt{ eq } \mathcal{L}_n$ for some $c \in \mathbf{N}^+$. Note that using standard semantics, we are free to select the order in which rules are applied. $S(n)$ consists of three phases where rule applications double the number of non-catalyst constraints.

Phase 1

Application (for c times) of `confrontation` between $X \texttt{ eq } \mathcal{U}_n$ and $X \texttt{ eq } \mathcal{L}_n$:

$$X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n \mapsto_{\texttt{confront.}} X \texttt{ eq } \mathcal{U}_n \wedge \mathcal{U}_n \texttt{ eq } \mathcal{L}_n$$

Note that a copy of the constraint $X \texttt{ eq } \mathcal{L}_n$ remains unchanged. The number of constraints is unchanged in Phase 1.

Phase 2

Application (for c times) of `confrontation` between $X \texttt{ eq } \mathcal{L}_n$ and $X \texttt{ eq } \mathcal{U}_n$:

$$X \texttt{ eq } \mathcal{L}_n \wedge X \texttt{ eq } \mathcal{U}_n \mapsto_{\texttt{confront.}} X \texttt{ eq } \mathcal{L}_n \wedge \mathcal{L}_n \texttt{ eq } \mathcal{U}_n$$

The number of constraints is unchanged in Phase 2.

Phase 3

Application of `decomposition` and `orientation` to c copies of $\mathcal{L}_n \texttt{ eq } \mathcal{U}_n$ and to c copies of $\mathcal{U}_n \texttt{ eq } \mathcal{L}_n$:

$$\mathcal{L}_n \texttt{ eq } \mathcal{U}_n \mapsto_{\texttt{decomp.}}\mapsto^* X \texttt{ eq } \mathcal{U}_{n-1} \wedge X \texttt{ eq } \mathcal{L}_{n-1}$$
$$\mathcal{U}_n \texttt{ eq } \mathcal{L}_n \mapsto_{\texttt{decomp.}}\mapsto^* X \texttt{ eq } \mathcal{U}_{n-1} \wedge X \texttt{ eq } \mathcal{L}_{n-1}$$

Each of the application removes *one* equation while producing *two* new equations. The number of non-catalyst constraints doubles in Phase 3.

Example 3. The computation steps of sub-computation $S(2)$ applied on the query $\left(\bigwedge_{i=1}^{2} X \texttt{ eq } \mathcal{L}_i\right) \wedge X \texttt{ eq } \mathcal{U}_2 \wedge X \texttt{ eq } \mathcal{L}_2$ are given given in Fig. 2. The *catalyst part* $\bigwedge_{i=1}^{2} X \texttt{ eq } \mathcal{L}_i$ (the first two constraints in each state) is unchanged. If we run sub-computation $S(1)$ on the answer given by $S(2)$, we first apply rule `confrontation` for four times on copies of $X \texttt{ eq } f(X,X) \wedge X \texttt{ eq } f(X,X)$. Finally, the generated four copies of $f(X,X) \texttt{ eq } f(X,X)$ are simplified to eight copies of $X \texttt{ eq } X$ by repeated application of `decomposition`.

[5] Remember that CHR conjunctions are considered as *multi-sets* of atomic constraints.

        `X eq f(X,X), X eq f(X,f(X,X)), X eq f(f(X,X),X), X eq f(X,f(X,X))`
$\mapsto_{\mathsf{co.}}$ `X eq f(X,X), X eq f(X,f(X,X)), X eq f(f(X,X),X), f(f(X,X),X) eq f(X,f(X,X))`
$\mapsto_{\mathsf{co.}}$ `X eq f(X,X), X eq f(X,f(X,X)), f(X,f(X,X)) eq f(f(X,X),X), f(f(X,X),X) eq f(X,f(X,X))`
$\mapsto_{\mathsf{de.}}\mapsto^{*}$ `X eq f(X,X), X eq f(X,f(X,X)), X eq f(X,X), X eq f(X,X), f(f(X,X),X) eq f(X,f(X,X))`
$\mapsto_{\mathsf{de.}}\mapsto^{*}$ `X eq f(X,X), X eq f(X,f(X,X)), X eq f(X,X), X eq f(X,X), X eq f(X,X), X eq f(X,X)`

Fig. 2. Sub-computation $S(2)$ applied on $\left(\bigwedge_{i=1}^{2} X \texttt{ eq } \mathcal{L}_i\right) \wedge X \texttt{ eq } \mathcal{U}_2 \wedge X \texttt{ eq } \mathcal{L}_2$

Lemma 6 *Application of* $S(n)$ *replaces* each copy *of the conjunction* $X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n$ *by* two copies *of the conjunction* $X \texttt{ eq } \mathcal{U}_{n-1} \wedge X \texttt{ eq } \mathcal{L}_{n-1}$ *for* $n \in \mathbf{N}^+$. *The catalyst constraint* $X \texttt{ eq } \mathcal{L}_n$ *remains unchanged while the number of rewritten non-catalyst equations* doubles*.*

Proof. We apply sub-computation $S(n)$ using the *catalyst* copy of $X \texttt{ eq } \mathcal{L}_n$ on the c copies of the conjunction $X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n$. Applying rules according to the phases of the $S(n)$ we create $2c$ copies of the conjunction $X \texttt{ eq } \mathcal{U}_{n-1} \wedge X \texttt{ eq } \mathcal{L}_{n-1}$. □

Proof (Lemma 5). We apply $S(i)$ repeatedly starting with the initial query $(\bigwedge_{i=1}^{n} X \texttt{ eq } \mathcal{L}_i) \wedge X \texttt{ eq } \mathcal{U}_n \wedge X \texttt{ eq } \mathcal{L}_n$. Formally, we use induction on the sequential application of $S(n), S(n-1), \ldots, S(1)$ which is possible because the *catalyst part* $\bigwedge_{i=1}^{n} X \texttt{ eq } \mathcal{L}_i$ remains unchanged. Doubling the number of copies of the rewritten (non-catalyst) constraints each time we apply $S(i)$, we arrive at 2^{n+1} (non-catalyst) constraints `X eq X`. □

The sub-computation $S(n)$ of the exponential query crucially relies on a scheduling of the rules such that all the phases are possible. We can simulate this instance of standard operational semantics by a CHR program for refined semantics (sources available [1]). When we improve the complexity of the solver to quadratic in Section 5, we will see that such a scheduling is not possible for flat constraints. Actually it seems that it is impossible under the refined semantics, thus impossible in any practical sequential implementation of CHR that currently exists. Thus the worst-case complexity of the RT solver (instantiated with a corrected order) for refined semantics is still an open problem.

4.5 Worst-Case Time and Space Complexity

Combining our results from the preceeding two subsections, we can now give our main result: The RT solver with measure order $\prec_m$ has exponential space and (hence) exponential time complexity under the standard semantics of CHR.

Lemma 7 *Any conjunction of equations* C *with size* $\#C$ *has problem measure* $|C| = O(2^{\#C})$.

Proof. Skipped for lack of space. □

As the problem measure is (at most) exponential in the size of the problem, the derivation length is (at most) exponential (by Lemma 3). We constructed a query of problem size $O(n^2)$ which produces more than 2^n equations for a specific computation strategy under the standard semantics. Therefore, our bound for the derivation length and the resulting exponential worst-case complexity are tight.

Theorem 1 (Exponential Complexity) *For the RT solver of Fig. 1 with measure order $\prec_m$, the number of rule applications is exponential in the size of the problem in the worst-case.*

Proof. By Lemma 3 and Lemma 7, the number of rule applications is at most exponential in the problem size. By Lemma 5 this upper bound is tight. □

As $\mathcal{L}_n \preceq_s \mathcal{U}_n$ and $\mathcal{U}_n \preceq_s \mathcal{L}_n$ in term-size order, $S(i)$ is also applicable and the solver with term-size order has *at least* exponential worst-case complexity under the standard semantics.

5 Quadratic Complexity

We can improve the worst-case time and space complexity of the CHR rational tree solver from exponential to quadratic by simply requiring that equations are in flat normal form when the problem is given. A term can be flattened by performing the opposite of variable elimination. Each sub-term is replaced by a new variable that is equated with the replaced expression.

Definition 9. *A conjunction of constraints is in* flat normal form *if each argument of each constraint contains at most one function symbol, i.e., it is either a variable or a function applied to variables.*

For flattening it suffices to traverse the constraints of the problem once and to replace nested function symbols by a new variable and a new equation with that variable. (A function symbol is *nested* if it occurs inside another term.) For our proofs it is not necessary that the flattening function produces the minimal number of equations.

Definition 10. *The* flattening function $[.]$ *transforms the syntactic equality constraints into an equivalent conjunction of flattened equations. For a conjunction of constraints $\bigwedge_{i=1}^{n} S_i \ \mathtt{eq}\ T_i$, we introduce new variables $X_1, \ldots, X_n$ and define*

$$\left[\bigwedge_{i=1}^{n} S_i \ \mathtt{eq}\ T_i\right] := \bigwedge_{i=1}^{n} \big([X_i \ \mathtt{eq}\ S_i]_1 \wedge [X_i \ \mathtt{eq}\ T_i]_1\big)$$

For an atomic constraint $X \ \mathtt{eq}\ T$, we define the auxiliary function $[.]_1$ as follows (with new variables $X_1, \ldots, X_n$)

$$[X \ \mathtt{eq}\ T]_1 := \begin{cases} X \ \mathtt{eq}\ T & \text{if } \mathrm{var}(T) \\ X \ \mathtt{eq}\ f(X_1, \ldots, X_n) \wedge \big(\bigwedge_{i=1}^{n} [X_i \ \mathtt{eq}\ T_i]_1\big) & \text{if } T = f(T_1, \ldots, T_n) \end{cases}$$

Lemma 8 *The size of the flattened problem $\#[C]$ is linear in the problem size, i.e., $\#[C] = O(\#C)$. Also the number of new variables and the number of new equations is linear in the problem size.*

Proof. ¿From Definition 10 we can see that the variables and function symbols of the original problem are kept. In addition, new variables are introduced for each original equation and for each nested function symbol and variable. Each new variable occurs twice. The number of original equations and of nested function symbols and variables is bounded by the number of function symbols and variables in the problem, i.e. by the problem size, because the arguments of an equation are not-nested symbols, either variables or outermost function symbols.

Therefore the size of the flattened problem is at most three times the size of the original problem size. Also, the number of new variables is bounded by the problem size, and thus the number of variables is linear in the problem size. Since the flattened equations have a new variable as first argument and each original equation is replaced by two new ones, the number of equations is at most twice the problem size. □

Lemma 9 *The flattening of a problem C can be done in linear time and space w.r.t. the problem size $\#C$.*

Proof. By Lemma 8 and by Definition 10 of the flattening function [.]. □

In flattened problems, the problem measure is quadratic in the problem size, while for general problems, it was exponential. The improvement is due to the fact that the depth of flat terms is at most one.

Lemma 10 *Given a* flat *problem C with at least one variable, its measure $|C| = O(v\,\#C)$ is bounded by the problem size and number of variables v.*

Proof. Analogous to the problem size, the problem measure is defined as the sum of the measures of the atomic constraints' arguments in the problem. Therefore a case analysis on the structure of flat terms suffices.

1. For any variable $|X_i| = i \leq v\,\#X_i$ as $i \leq v$ and $\#X_i = 1$.
2. For any flat function term $T = f(X_{j_1}, \ldots, X_{j_n})$, we have $|T| = n + 2\sum_{j=1}^{n} j_i \leq n + 2\sum_{j=1}^{n} v = (2v+1)n \leq (2v+1)\,\#T$ as $\#T = n+1$.

By Definitions 5 and 6 we conclude $|C| = O(v\,\#C)$. □

Now we can prove that the overall complexity is quadratic in the problem size when the problem is in flat normal form.

Theorem 2 (Quadratic Complexity) *For the RT solver of Fig. 1 with measure order $\prec_m$, the number of rule applications is quadratic in the problem size in the worst-case if the problem is in flat normal form.*

Proof. The number of rule applications is bounded by the problem measure by Theorem 1. Given a problem C, by Lemmas 8 and 10, the measure size of the problem in flat normal form is quadratic in the problem sizes: $\big|[C]\big| = O(v\,\#[C]) = O(\#^2[C]) = O(\#^2 C)$. □

Analogous proofs of quadratic time and space complexity can be given for terms with *bounded depth*, only the constant factors increase.

For flattened problems, the rule `decomposition` either fails due to a clash or produces equations between variables only. Flat terms that do not clash have the same term size. So it does not matter how function terms of same size are ordered by the instance of the generic term order $\prec$. Therefore Theorem 2 is also applicable to any instance of the generic term order used in the RT solver. We conjecture that even the Prolog built-in term order `@<` is sufficient, since functional terms can be ordered arbitrarily without changing the sizes of the equations involved.

6 Conclusion

The complexity of the classic CHR rational tree equation solver [7, 4, 8, 14] was an open problem. For termination, the solver relies on a generic order between terms that must fulfil some properties. The standard implementation of the solver that is included in many CHR libraries uses the built-in Prolog term order that does not respect all properties. We gave an example for non-termination of that solver.

Our main result shows that there exists a term order for the classic CHR rational tree equation solver that leads to *exponential worst-case time and space complexity* in the size of the problem under the standard CHR semantics (that does not constrain the order of rule applications). This complexity bound is *tight*. This term order, however, is not an instance of the generic term order. It is based on a term measure that is exponential in the depth of the term.

Since the generic term order usually required for termination of the RT solver and the measure order we have defined in this paper are incompatible, we conjecture that there is a more general generic order that subsumes these orders, and that this order is based on the sub-term relation. We are also interested in other measures based on term-depth or explicit exponentiation like $2^{\#T}$.

Our complexity proof does not apply to actual implementations of the RT solver that usually rely on the refined CHR semantics, but it implies that under standard semantics, their complexity is *at least* exponential when the term-size instance of the generic term order is used. It is still an open question, whether the complexity of the solver with generic term order using other order instances and/or using the refined semantics is polynomial or not. However, this question is not so burning anymore in the light of our following result.

We improved the complexity of the solver to be *quadratic for any term order* (including the built-in Prolog one) under *standard and refined semantics* by simply requiring that equations are in flat normal form before solving them. Since any conjunction of equations can be flattened in linear time and space, this gives an efficient polynomial algorithm.

Since there is no performance penalty in time and space complexity (except constant factors) when using CHR [16], one may be interested in a *quasi-linear* solver. Such a solver is implementable by a straightforward combination [2] of the RT solver with the union-find algorithm in CHR [15, 6] that will handle all equations between variables. During this line of work, the CHR RT solver should be more thoroughly compared to existing classical unification algorithms (c.f. Section 1). This includes to check if our proof methods apply to implementations of these algorithms as well.

CHR program sources for this paper are available [1].

References

1. CHR program sources for this paper (SICStus Prolog 3.11), 2006. `http://www.informatik.uni-ulm.de/pm/index.php?id=133`.
2. S. Abdennadher and T. Frühwirth. Integration and optimization of rule-based constraint solvers. In M. Bruynooghe, editor, *LOPSTR 2003*, volume 3018 of *LNCS*, pages 198–213. Springer, 2003.
3. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, London, 1982.
4. T. Frühwirth. Theory and practice of Constraint Handling Rules, Special issue on constraint logic programming. *Journal of Logic Programming*, 37(1-3):95–138, 1998.
5. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 547–557. Morgan Kaufmann, 2002.
6. T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.
7. T. Frühwirth and S. Abdennadher. *Constraint-Programmierung*. Springer, 1997.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. J. Herbrand. *Recherches sur la théorie de la demonstration*. PhD thesis, Université de Paris, France, 1930.
10. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.
11. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.
12. M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.
13. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
14. T. Schrijvers and T. Frühwirth. Constraint Handling Rules (CHR) web page, 2006. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
15. T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules, programming pearl. *Theory and Practice of Logic Programming (TPLP)*, 6(1&2):213–224, 2006.
16. J. Sneyers, T. Schrijvers, and B. Demoen. The computational power and complexity of Constraint Handling Rules. In T. Schrijvers and T. Frühwirth, editors, *CHR 2005*, pages 3–18, Sitges, Spain, 2005. Report CW421, K.U. Leuven, Belgium.
17. R. E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.