As Time Goes By: Automatic Complexity Analysis of Simplification Rules

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

Abstract

From a suitable termination order, called a tight ranking, we can automatically compute the worst-case time complexity of a CHR constraint simplification rule program from its program text: We combine the worst-case derivation length of a query predicted from its ranking with a worst-case estimate of the number and cost of rule application attempts and the cost of rule applications to obtain the desired meta-theorem. For two Boolean and a path consistency solver, the predictions are computed and are compared to some empirical run-time measurements.

1 Introduction

The programming language *CHR (Constraint Handling Rules)* [Fru98] was originally introduced to ease the development of constraint solvers. Currently several CHR libraries exist in languages such as Prolog, Haskell and Java, and dozens of projects use CHR. Over time it has become apparent that CHR and its extensions [Abd00] are useful for implementing reasoning systems in general, including deduction and abduction, since techniques like forward and backward chaining, bottom-up and top-down evaluation, integrity constraints, tabulation/memoization can be easily implemented and combined [AAI00, SaAb00].

CHR are a committed-choice concurrent constraint logic programming language consisting of guarded rules that work on conjunctions of constraints. A CHR program consists of simplification and propagation rules. Simplification replaces constraints by simpler constraints while preserving logical equivalence. Propagation adds new constraints which are logically redundant but may cause further simplification.

Properties like rule confluence [AFM99] and program equivalence [AbFr99] have been investigated for CHR. These properties are decidable for terminating programs. In a previous paper [Fru00a] we have proven termination of simplification rule programs using rankings. A ranking maps lhs (left hand side) and rhs (right hand side) of each simplification rule to a natural number, such that the rank of the lhs is strictly larger than the rank of the rhs. 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 [Fru00b], because each rule application decreases the rank by at least one.

Example 1.1 *Consider the constraint* `even` *that ensures that a natural number (written in successor notation) is even:*

```
even(0) <=> true.
even(s(N)) <=> N=s(M), even(M).
```

The first rule says that `even(0)` *can be simplified to* `true`*, a built-in constraint that is always satisfiable. In the second rule, the built-in constraint* `=` *stands for syntactic equality:* `N=s(M)` *ensures that* `N` *is the successor of some number* `M`*. The comma stands for conjunction* $\wedge$*. 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.*

If a constraint matches the lhs of a rule, it is replaced by the rhs of the rule. If no rule matches a constraint, the constraint delays. For example, the query `even(N)` *delays. When the constraint* `N=0` *is added,* `even(N)` *is woken and behaves like the query* `even(0)`*. It reduces to* `true` *with the first rule. To the query* `even(s(X))` *the second rule is applicable, the answer is* `X=s(M),even(M)`*. The query* `even(s(0))` *results in an inconsistency after application of the second rule, since* `0=s(M)` *is unsatisfiable.*

An obvious ranking for the rules of `even` *is*

$$rank(even(N)) = size(N)$$

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

The ranking not only proves termination, it also gives an upper bound on the derivation length, in case the argument of `even` *is completely known (ground): With each rule application, we decrease the rank of the argument of* `even` *by* 2*.*

In [Fru00b] we have shown that the derivation length is not a suitable measure for time complexity. The runtime of a CHR program not only depends on the number of rule applications, but also, more significantly, on the number of rule application *attempts* (rule tries).

In this paper we combine the predicted worst-case derivation length with a worst-case estimate of the number and cost of rule tries and the cost of rule applications to obtain a meta-theorem for the worst-case time complexity of CHR constraint simplification rule programs. In the theorem, we will make no specific assumptions on the implementation of CHR, so it applies to naive implementations of CHR as well.

Example 1.2 *[Contd.] It is easy to show that the worst-case time complexity of a single* `even` *constraint is linear in the derivation length, i.e. the rank. The same observation holds for a query consisting of several ground* `even` *constraints, if the rank is defined as the sum of the ranks of the individual constraints.*

However, things change when we add the rule:

```
even(s(X)),even(X) <=> false.
```

where `false` *is a built-in constraint that is always unsatisfiable. This rule may be applicable to* all pairs of `even` *constraints in a query, and again after a reduction of a single* `even` *constraint with one of the other two rules. Of course in most cases, the rule application attempts (rule tries) will be in vain.*

Thus the number of rule tries in a single derivation step is quadratic in the number of `even` *constraints in the query. Since the rank of an* `even` *constraint is at least one, the rank of the query is a bound on the number of constraints. The number of derivation steps is also bounded by the rank of the query. Overall, this yields an algorithm that is cubic in the rank of the query.*

Related Work. To the best of our knowledge, only [McA99, GaMc01] are closely related to our work in that they give several complexity meta-theorems for a logical rule-based language. These papers investigate bottom-up logic programming as a formalism for expressing static analyses and related algorithms. [McA99] is concerned with certain propagation rules (in our terminology), while [GaMc01] extends the rule language with deletions of atomic formulae and static priorities between rules. Such rules correspond to CHR simplification or simpagation rules [Fru98] that are applied in textual order.

McAllester and Ganzinger prove several complexity theorems which allow, in many cases, to determine the asymptotic running time of a bottom-up logic programm by inspection. The main difference and complementarity between their work and our paper is that they consider rules that must be applied to ground formulae at run-time, while we consider simplification rules that involve free variables at run-time and arbitrary built-in constraints. They deal with the complexity of optimally implemented programs using clever indexing and structure sharing, while our results apply also to naive implementations of CHR.

In the complexity meta-theorem of [GaMc01], the complexity is the sum of the syntactic size of the query and the worst number of potential prefix firings (ground sub-formulas of lhs instances that could occur in a derivation) of the rules in the program. Here it is - ignoring the cost of built-in constraints - the sum of the rank of the query and the number of potential rule applications. The computation of the number of prefix firings requires insights about the states of all valid computations that can be performed. The number of potential rule applications can be computed automatically from the program text, once a ranking is known.

Overview of the Paper. We will first give syntax and semantics of CHR. In Section 3, we introduce rankings and show how they can be used to derive tight upper bounds for worst-case derivation lengths. In the next section we show how to use these derivation lengths to predict the worst-case complexity of CHR programs. Finally, the fifth section reviews some CHR constraint solver programs. Based on the predicted worst-case derivation lengths, the worst-case time complexity is computed according to our complexity meta-theorem. The prediction is compared with empirical run-time measurements. We conclude with a discussion of the results obtained. This paper is a revised version of a paper presented at a non-archival workshop [Fru01]. We apply our complexity theorem to two Boolean constraint solvers and one path consistency solver. Preliminary empirical results indicate that the actual run-time of the solvers in the Sicstus Prolog implementation of CHR is better due to CHR compiler optimizations like indexing.

2 Syntax and Semantics

In this section we give syntax and semantics for CHR, for details see [AFM99]. We assume some familiarity with (concurrent) constraint (logic) programming [JaMa94, MaSt98].

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

In the following definitions, 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*

$$n \ @ \ H \ \texttt{<=>} \ G \mid B$$

and a propagation CHR *is of the form*

$$n \ @ \ H \ \texttt{==>} \ G \mid B$$

where the rule has an optional name `n` *followed by the symbol @. The lhs H (head) is a conjunction of CHR constraints. The optional guard G followed by the symbol | is a conjunction of built-in constraints. The rhs B (body) is a conjunction of built-in and CHR constraints.*

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.

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. A* derivation *is a sequence of derivation steps $S_1 \longmapsto S_2 \longmapsto S_3 \ldots$. The* derivation length *is the number of derivation steps in a derivation.*

Definition 2.3 *Let P be a CHR program and CT be a constraint theory for the built-in constraints. The transition relation $\longmapsto$ for CHR is as follows:*

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

Propagate
$H' \wedge C \longmapsto (H = H') \wedge G \wedge B \wedge H' \wedge C$
if $(H \ \texttt{==>} \ G \mid B)$ *in* P *and* $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$

When we use a rule from the program, we will rename its variables using new symbols, and these variables are denoted by the sequence $\bar{x}$. A rule with lhs H and guard G is *applicable* to CHR constraints H' in the context of constraints C, when the condition holds that $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$.

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 the rhs B to the state and also adds the equation $H = H'$ and the guard G. If a propagation rule (H `==>` G | B) is applied to H', the **Propagate** transition adds B, $H = H'$ and G, but does not remove H'. Trivial non-termination is avoided by applying a propagation rule at most once to the same constraints [Abd97].

We finally discuss in more detail *the rule applicability condition* $CT \models C \rightarrow \exists \bar{x}(H = H' \wedge G)$. The equation $(H = H')$ is a notational shorthand for equating the arguments of the CHR constraints that occur in H and H'. More precisely, by $(H_1 \wedge \ldots \wedge H_n) = (H'_1 \wedge \ldots \wedge H'_n)$, where conjuncts can be permuted, we mean $(H_1 = H'_1) \wedge \ldots \wedge (H_n = H'_n)$. By equating two constraints, $c(t_1, \ldots, t_n) = c(s_1, \ldots, s_n)$, we mean $(t_1 = s_1) \wedge \ldots \wedge (t_n = s_n)$. The symbol $=$ is to be understood as built-in constraint for syntactic equality.

Operationally, the rule applicability condition can be checked as follows: Given the built-in constraints of C, try to solve the built-in constraints $(H = H' \wedge G)$ without further constraining any variable in H' and C. This means that we first check that H' matches H and then check the guard G under this matching.

As a consequence, in a CHR implementation, there are several computational phases when a rule is applied:

LHS Matching: Atomic CHR constraints in the current state have to be found that match the lhs constraints of the rule.

Guard Checking: It has to be checked if the current built-in constraints imply the guard of the rule.

RHS Handling: The built-in and CHR constraints of the rhs are added. Before that, the CHR constraints of the lhs are removed.

In this paper we are only concerned with simplification rules. For the rest of the paper we assume that CHR programs do not contain any propagation rules.

3 Rankings for Derivation Lengths

In this section, we introduce rankings for constraint simplification rules and show how the rankings can be used to derive tight upper bounds for worst-case derivation lengths of CHR programs.

3.1 Rankings

A ranking maps terms and formulae to natural numbers.

Definition 3.1 *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* (linear polynomial) CHR ranking (function) *defines the rank of a term or constraint 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 $rank(X) \geq 0$. For each built-in constraint C we impose $rank(C) = 0$. For each CHR constraint D we impose $rank(D) > 0$.

The rank of a conjunction is the sum of the ranks of its conjuncts:

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

The rank of any built-in constraint is 0, since we assume that they always terminate and since their derivation length is 0. A built-in constraint may imply order constraints between the ranks of its arguments (interargument relations), such as $s = t \rightarrow rank(s) = rank(t)$.

In [Fru00a] we prove termination for CHR simplification rule programs under any scheduling of rule applications using CHR rankings. We want to make sure that the rank of the lhs of a rule is always strictly larger than the rank of the rhs of the rule.

Definition 3.2 *Let rank be a CHR ranking function. The* ranking (condition) *of a simplification rule H `<=>` $G \mid C \wedge B$, where C are built-in constraints and B are CHR constraints, is the formula*

$$\forall\ (O \rightarrow rank(H) > rank(B)),$$

where O is a conjunction of order constraints implied by the built-in constraints in the rule, $\forall\ ((G \wedge C) \rightarrow O)$.

Implied order constraints such as O help to establish that rank of the lhs of the rule is strictly larger than the rank of the rhs. Since termination is undecidable for CHR, a suitable ranking and suitable order constraints cannot be found completely automatically.

3.2 Derivation Lengths

A ranking of a program satisfying the ranking condition for each rule ensures that the rank of the lhs of each rule is strictly larger than the rank of its rhs. Intuitively then, the rank of a query gives us an upper bound on the number of rule applications (derivation steps), i.e. derivation lengths. In [Fru00b] we predicted worst-case derivation lengths using rankings and compared the predictions with empirical data.

Definition 3.3 *A goal G is* bounded *if the rank of any instance of G is bounded from above by a constant.*

Theorem 3.1 *Let P be a CHR program containing only simplification rules.*

1. [Fru00a] If the ranking condition holds for each rule in P, then P is terminating for all bounded goals.

2. [Fru00b] If the ranking condition holds for each rule in P, then a worst-case derivation length D for a bounded goal G in P is the rank of G:

$$D = rank(G)$$

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

Definition 3.4 *Let S be a* ranking of a simplification rule *of the form H `<=>` $G \mid B$. The ranking is* exact for the rule S *iff* $rank(H) = rank(B) + 1$. *The ranking is* tight by n for the rule S *iff* $rank(H) = rank(B) + n$. *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.*

4 Worst-Case Time Complexity

We first consider the worst cost of applying a single rule, which consists of the cost to try the rule on all constraints in the current state and of the cost to apply the rule to some constraints in the state. Then we choose the worst rule in the program and apply it in the worst possible state of the derivation. Multiplying the result with the worst-case derivation length gives us the desired upper bound on the worst-case time complexity.

In the following, we assume a naive implementation of CHR with no optimizations. The complexity of handling built-in constraints is predetermined by the built-in constraint solvers used. We assume that the time complexity of checking and adding built-in constraints is not dependent on the constraints accumulated so far in the derivation.

Lemma 4.1 *Let S be a simplification rule of the form H* `<=>` *$G \mid C \wedge B$, where H is a conjunction of n CHR constraints, G and C are built-in constraints and B are CHR constraints. The* worst-case time complexity of applying the rule S *in a state with c CHR constraints is:*

$$O(c^n(O_H + O_G) \; + \; (O_C + O_B)),$$

where O_H is the complexity of matching the lhs H of the rule, O_G the complexity of checking the guard G, O_C the complexity of adding the rhs built-in constraints C, and O_B the complexity of removing the lhs CHR constraints and of adding the rhs CHR constraints B.

Proof. *The formula consists of two summands, the first is the cost of trying the rule, the second the cost of applying the rule. In a naive implementation, we compute all possible combinations of n constraints and try to match them to the lhs of the rule. Hence, given c constraints in a query and a rule with n lhs constraints, there are $O(c^n)$ combinations of constraints to try. Each try involves matching the lhs of the rule with complexity O_H and, in the worst-case, checking the guard with complexity O_G. In the worst-case, all possible combinations have been tried before the rule is finally applied. Then, the cost of handling the rhs of the rule, $(O_C + O_B)$, is incurred.*

Now we are ready to give our meta-theorem about the time complexity of simplification rule programs. To compute the time complexity of a derivation, we have to find the worst-case for the application of a rule, i.e. the largest number of CHR constraints c_{max} of any state in a derivation and the most costly rule that could be tried and applied. We know that the number of derivation steps is bounded by the rank D. It turns out that D is also an upper bound for c_{max}.

Theorem 4.1 *Let P be a CHR program containing only constraint simplification rules. Let Q be a query with worst-case derivation length D. Then the worst-case time complexity of a derivation starting with Q is:*

$$O(D \sum_i (D^{n_i}(O_{H_i} + O_{G_i}) \; + \; (O_{C_i} + O_{B_i}))),$$

where the index i ranges over the constraint simplification rules in the program P.

Proof. *In the worst-case of a naive implementation, in each of the D derivation steps, all rules are tried on all combinations of the maximum possible number of constraints c_{max} and then the most costly rule is applied. Since rule application attempts are independent from each other, we can extend Lemma 4.1 to a set of rules in a straightforward way:*

$$O(\sum_i c_{max}^{n_i}(O_{H_i} + O_{G_i}) \; + \; Max_i(O_{C_i} + O_{B_i})),$$

where c_{max} is the worst number of CHR constraints in a derivation from a given query and Max_i takes the maximum over all i. Since the function Max and $+$ are equivalent in the O-notation, we can replace Max_i by $\sum_i$. Multiplying the resulting formula by the derivation length D yields the overall complexity:

$$O(D \sum_i (c_{max}^{n_i}(O_{H_i} + O_{G_i}) \; + \; (O_{C_i} + O_{B_i}))).$$

There cannot be more than D CHR constraints in any state of a derivation starting with a query with worst-case derivation length D, because each CHR constraint has a rank of at least 1 by definition and because each derivation step decreases the value of D by at least 1. Thus c_{max} is bounded by D. After replacing c_{max} by the bound D, we arrive at the formula of the Theorem.

From the meta-theorem it can be seen that the cost of rule tries dominates the complexity of a naive implementation of CHR.

We end this section with some general remarks on the complexities of the constituents of a simplification rule. The cost of syntactic matching O_H is determined by the syntactic size of the lhs in the given program text. Thus, its time complexity is constant.

The complexity of guard checking O_G is usually as most as high as the complexity of adding the respective constraints. The worst-case time complexity of adding built-in constraints O_C is typically linear in their size.

We assume that the complexity O_B of removing and adding CHR constraints (without applying any rules) is constant in a naive implementation where e.g. lists are used to store the CHR constraints.

5 Time Complexity of CHR Constraint Solvers

We now derive worst-case time complexities of two Boolean and one path consistency CHR constraint solver [Fru98] from the CHR library of Sicstus Prolog [HoFr98, HoFr00]. We will use *concrete syntax* of Prolog implementations of CHR, where a conjunction is a sequence of conjuncts separated by commas.

We will contrast these results with the time complexities derived from a set of test-runs with randomized data. We expect the empirical results to be better than the predicted ones, since this CHR implementation uses indexing for computing the combinations of constraints needed for lhs matching of a rule.

The Sicstus Prolog and CHR source code for the test-runs is available at `www.informatik.uni-muenchen.de/~fruehwir/chr/complexity.pl`. The code can be run via the WWW-interface of CHR Online [SaAb00].

For each solver, we will give a ranking that is an upper bound on the derivation length. From the ranking, we calculate the worst-case time complexity. We denote constant time complexity by the number 1 and zero time by 0 (this means that no computation is performed at all). We will summarize the empirical results of the test-runs in a table, see e.g. Figure 1. The tables have the following columns:

Goal the (abbreviated) goal that was run to produce the test data.

Worst the predicted worst-case derivation length D for the goal.

Apply the actual number of rule applications, i.e. derivation length.

Try the number of rules that have been tried, but not necessarily applied.

Time the time to run the goal with the CHR libarary of Sicstus Prolog, in seconds, including instrumented source code for randomization, on a recent Linux PC with medium work load.

5.1 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, which are modeled here as CHR constraints. Syntactic equality `=` is a built-in constraint. In the constraint solver *Bool*, we simplify a single constraint $and(X, Y, X \wedge Y)$ into one or more equations whenever possible:

```
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 goal `and(X,Y,Z),X=0` will result in `X=0, Z=0`.

Derivation Length. Since a single rule application reduces each CHR constraint to built-in constraints, the worst-case derivation length is just the number of constraints in the query, c. Let the ranking be

$$rank(A) = 1 \text{ if } A \text{ is an atomic CHR constraint}$$

For each rule in *Bool*, H `<=>` $G \mid B$, 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 goal is

$$D_{Bool} = c$$

It can be much smaller. For example, the goal `and(U,V,W)` delays, its derivation length is zero. Another example is a goal 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.

Complexity. All rules have one lhs CHR constraint, i.e.. $n = 1$. The derivation length is bounded by c. Checking or establishing built-in syntactic equality between variables and the constants `0` and `1` can be implemented in constant time. Then, for all rules, (O_H, O_G, O_C, O_B) is $(1, 1, 1, 0)$, i.e. all rule-dependent complexities are constant. According to the complexity meta-theorem, this gives $O_{Bool}(c(c^1(1+1)+(1+0)))$, i.e

$$O_{Bool}(c^2)$$

Empirical Results. In Figure 1, the Prolog predicate `tst/3` produces a chain of `and` constraints, where the last variable of one constraint is the first variable of the next constraint. The first (`A`) and the last (`B`) variable are returned in the second and third argument of `tst/3`, respectively.

Figure 1 shows that

- The order of (built-in) constraints may strongly influence the run-time.

Goal	Worst	Apply	Try	Time
tst(125,A,B), A=1	125	1	759	0.06
tst(250,A,B), A=1	250	1	1509	0.10
tst(500,A,B), A=1	500	1	3009	0.22
tst(1000,A,B), A=1	1000	1	6009	0.43
tst(2000,A,B), A=1	2000	1	12009	0.87
tst(4000,A,B), A=1	4000	1	24009	1.73
tst(8000,A,B), A=1	8000	1	48009	3.46
tst(125,A,B), B=1	125	125	1500	0.11
tst(250,A,B), B=1	250	250	3000	0.24
tst(500,A,B), B=1	500	500	6000	0.47
tst(1000,A,B), B=1	1000	1000	12000	0.95
tst(2000,A,B), B=1	2000	2000	24000	1.88
tst(4000,A,B), B=1	4000	4000	48000	3.75
tst(8000,A,B), B=1	8000	8000	96000	7.51
tst(125,A,B), A=0	125	125	875	0.07
tst(250,A,B), A=0	250	250	1750	0.15
tst(500,A,B), A=0	500	500	3500	0.29
tst(1000,A,B), A=0	1000	1000	7000	0.57
tst(2000,A,B), A=0	2000	2000	14000	1.16
tst(4000,A,B), A=0	4000	4000	28000	2.34
tst(8000,A,B), A=0	8000	8000	56000	4.67
A=0, tst(125,A,B)	125	125	125	0.01
A=0, tst(250,A,B)	250	250	250	0.02
A=0, tst(500,A,B)	500	500	500	0.03
A=0, tst(1000,A,B)	1000	1000	1000	0.06
A=0, tst(2000,A,B)	2000	2000	2000	0.11
A=0, tst(4000,A,B)	4000	4000	4000	0.21
A=0, tst(8000,A,B)	8000	8000	8000	0.44

Figure 1: Results from Test-Runs with Boolean And

- The actual derivation length reaches the predicted worst-case derivation length.
- The number of rule applications may be arbitrarily small.
- The number of rule tries is up to 12 times larger than the worst-case derivation length. Note that there are 6 rules.
- Run-time is linear in the number of rule tries.

These observations will also hold for the other constraint solvers we considered, except of course for the relationship between the number of rule applications and the number of rule tries.

In practice, the observed time complexity of the solver seems to be linear:

$$O^{obs}_{Bool}(c)$$

We attribute this difference to the effect of indexing on variables which allows to find matching constraints faster.

Boolean Cardinality

The cardinality constraint combinator was introduced in the CLP language cc(FD) [vHSD95] for finite domains. In the solver *Card* we adapted cardinality for Boolean variables. The Boolean cardinality constraint `#(L,U,BL,N)` is true if the number of Boolean variables in the list `BL` that are equal to `1` is between `L` and `U`. `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, 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 this CHR program, all constraints except cardinality are built-in. `all(B,L)` equates all elements of the list `L` to `B`. `delete(X,L,L1)` deletes the element `X` from the list `L` resulting in the list `L1`. Due to the semantics of guard checking, `X` must exactly match the element that is to be removed.

Derivation Length. Our ranking is based on the length of the list argument of the Boolean cardinality constraint:

$$rank(\#(L,U,BL,N)) = 1 + length(BL)$$
$$length([]) = 0$$
$$length([X|L]) = 1 + length(L)$$

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

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, the goal `#(0,0,[],0)` has derivation length one (no matter which of the three satisfaction rules is applied). Let the rank of atomic constraints in a query be bounded by a constant 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 ranking of the solver program *Card* is tight by l.

From the ranking we see that the derivation length of a single cardinality constraint is bounded by the length of the list argument. For example, the goal `#(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 goal is less or equal to the sum of the lengths of the lists occurring in the goal. Hence it is linear in the syntactic size of the goal in the worst-case.

Let c be the number of CHR constraints in a query and recall that the rank of atomic constraints is bounded by l. Then we have that:

$$D_{Card} = cl$$

Complexity. All rules have one lhs CHR constraint. Time complexity for the built-in constraints `delete` and `all` can be assumed to be linear in the length of the list, and is constant for the other built-in constraints. The derivation length is bounded by cl. The time complexities, (O_H, O_G, O_C, O_B), of the five rules are $(1,1,1,0), (1,1,l,0), (1,1,l,0), (1,l,1,1)$ and $(1,l,1,1)$ respectively. Hence the complexity for both the rule tries and the rule applications is at worst linear in l. According to the meta-theorem, the time complexity is $O(cl((cl)^1 l \; + \; l))$, i.e.

$$O(c^2 l^3).$$

Empirical Results. Our empirical results are presented in Figure 2. *allr* is a variation on *all*, it starts equating the list elements from the back of the list. This means that in the guards of the recursive rules for cardinality, `delete` has to search till the end of the list to find a zero or one. *card_rand* produces a random list of variables, zeros and ones, each of the three with the same probability. The list lengths were chosen at random between 0 and 1000 and then the problem instances were ordered by list length.

The table shows that

- For the `card_rand` problem, the figures follow from to the probability distribution.
- The other problem instances show the influence of the order of built-in constraints on the run-time. However, timings differ by a constant factor, so complexity is not affected.
- The number of rule tries is up to 10 times larger than the worst-case derivation length. Note that there are 5 rules, and they may be tried in vain.
- Run-time is roughly quadratic in the list length.

Goal	Worst	Apply	Try	Time
card_rand(N,A,B,L), #(A,B,L,N)	40	30	132	0.03
	91	56	260	0.12
	217	143	655	0.71
	298	199	901	1.25
	655	450	2029	5.07
	672	446	2008	5.10
#(0,1,L,N),all(0,L)	109	108	1071	0.65
	200	199	1981	1.98
	318	317	3161	4.01
	382	381	3801	5.27
#(0,1,L,N),allr(0,L)	109	108	1071	0.69
	200	199	1981	2.38
	318	317	3161	4.47
	382	381	3801	7.72
#(0,1,L,N),all(X,L), X=0	109	108	1076	0.34
	200	199	1986	0.99
	318	317	3166	2.29
	382	381	3806	3.21
all(0,L),#(0,1,L,N)	109	108	536	0.14
	200	199	991	0.45
	318	317	1581	1.12
	382	381	1901	1.58

Figure 2: Test-Run Results for Boolean Cardinality

We also did some experiments with more than one cardinality constraint but found that the overall run-time was the sum of the run-times of each constraint alone. Thus the observed time complexity has lower exponents than those predicted:

$$O^{obs}_{Card}(cl^2)$$

We again attribute this difference to the effect of indexing on variables.

5.2 Path Consistency

In this section we analyze a constraint solver that implements the classical artificial intelligence algorithm of path consistency [MaFr85, MoHe86].

A *disjunctive binary constraint* $c(I, J, \{r_1, \ldots, r_n\})$, also written $I \; \{r_1, \ldots, r_n\} \; J$, is a finite disjunction $(I \; r_1 \; J) \vee \ldots \vee (I \; r_n \; J)$, where each r_i is a binary relation. The r_i are called *primitive constraints.* The number p of primitive constraints is finite and they are pairwise disjoint.

W.l.o.g. we will assume that in a query, for each ordered pair of variables, there is a disjunctive binary constraint between them. The basic operation of path consistency computes a tighter constraint between two variables I and J by intersecting it with the constraint composed from the two constraints between I and a third variable K and between K and J. This operation can be implemented directly by a single rule in the solver *Path*:

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

The repeated application of the rule will make the initial query constraints path consistent. The built-in constraints `composition` and `intersection` implement functions on pairs of disjunctive binary constraints:

> $composition(C_1, C_2, C_3)$ iff
> $\quad I\ C_1\ K\ \wedge\ K\ C_2\ J \rightarrow I\ C_3\ J$,
> where C_3 is the smallest set of primitive constraints implied for given C_1 and C_2.
>
> $intersection(C_1, C_2, C_3)$ iff
> $\quad I\ C_1\ J\ \wedge\ I\ C_2\ J \leftrightarrow I\ C_3\ J$.

The check `C123=\=C3` makes sure that the new constraint `C123` is different from the old one `C3`.

Derivation Length. We rely on the following ranking:

$$rank(c(I,J,C)) = 1 + card(C)$$
$$card(\{c_1,\ldots,c_n\}) = n$$

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

For the ranking, one is added to the cardinality of C so that constraints with an empty set C have a positive rank as well. Queries are bounded, when C is known.

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 rhs is ranked strictly smaller than the lhs of the rule. 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`. Hence, if the maximum number of primitive constraints is p, the ranking is tight by p. The worst-case derivation length is linear in the syntactic size of the goal:

$$D_{Path} = cp$$

Complexity. There is one rule, it has three lhs CHR constraints. For small p, the built-in constraints for composition, intersection and inequality checking can be implemented by table look-up, i.e. in constant time. Otherwise, we define the operations in terms of primtive constraints. Composition of disjunctive constraints can be computed by pairwise composition of its primitive constraints. Intersection for disjunctive constraints can be implemented by set intersection, since primitive constraints are disjoint. We assume constant time access to individual elements in the composition table of primitive constraints. Then composition can be implemented in quadratic time, $O(p^2)$. Intersection and inequality checking can be implemented in linear time.

Hence, according to the meta-theorem, the complexity is $O(cp((cp)^3(1+(p^2+p+p))+(0+1)))$, i.e

$$O_{Path}(c^4p^6)$$

Empirical Results. In the goals of Figure 3, `tpath` generates constraints between each pair of different variables in its argument list. The disjunctive constraints C are randomly chosen non-empty subsets of $\{<,=,>\}$, each with the same probability. Hence p is a constant, $p = 3$. For a list of length n, there are exactly $c = n(n-1)$ constraints. Thus the worst case derivation length is $3n(n-1)$. The table entries have been sorted.

Goal	Worst	Apply	Try	Time
length(L,V),	168	32	1079	0.44
tpath(L,A),	168	41	1151	0.50
V=8	168	45	1477	0.65
	168	49	1279	0.55
..., V=12	396	87	4024	1.76
	396	101	4791	2.11
	396	102	4622	2.00
	396	104	4895	2.12
..., V=16	720	155	10241	4.54
	720	155	10724	4.82
	720	160	11709	5.23
	720	185	12330	5.54
..., V=20	1140	241	22075	10.33
	1140	263	23578	11.04
	1140	269	24154	11.24
	1140	277	23573	11.02

Figure 3: Test-Run Results for Path Consistency

The table shows that

- The actual derivation length is roughly linear in the predicted worst-case derivation length, i.e. linear in the number of constraints.
- The number of rule tries increases faster than the worst-case derivation length.

- Run-time is roughly linear in the number of rule tries. It is roughly cubic in the number of variables v.

We can conclude from the current experiments, where p is constant, that the observed complexity is much lower than the predicted one. Since $O(v^2) = O(c)$ we have:

$$O^{obs}_{Path}(v^3) = O^{obs}_{Path}(c^{1.5})$$

For $p = 3$, this corresponds to the complexity of the best known general algorithm for path consistency, which is $O(v^3p^3)$ [MaFr85, MoHe86].

6 Conclusions

Based on the worst-case derivation length, as given by a ranking, we were able to give a general complexity meta-theorem for the worst-case time complexity of CHR constraint simplification rule programs. Rankings were originally used to prove termination. They map constraints and terms to natural numbers such that the rank of the lhs of a rule is larger than the rank of the rhs of a rule. Once a ranking has been found, our meta-theorem allows for computing the complexity automatically from the program text. Our theorem also applies to naive implementations of CHR simplification rules.

We have found that the dominating factor in the complexity are the rule application attempts (rule tries), not the actual rule applications. The cost of rule tries depends on the number of lhs CHR constraints n, the complexity of the guard checking and the ranking D of a given query. D was bounded by the product cr, where c is the number of atomic CHR constraints in the query and r is the maximum rank of an atomic CHR constraint in the query. r often can be interpreted as syntactic size. Built-in constraints only contribute if they have non-constant complexity. This is the case if non-scalar datatypes like lists or sets are involved. In our examples, the derived complexities were of the form $c^{n+1}r^{n+1+k}$, where k is a small constant (often zero) introduced by the built-in constraints.

We compared the complexities predicted by our theorem with the complexities observed in empirical tests of two Boolean and a path consistency constraint solver. Due to optimizations like indexing on variables in the Sicstus Prolog CHR implementation, the observed complexities were better than the predicted ones.

They involved the same parameters, but lower exponents. In the case of the two Boolean constraint solvers, the complexity of rule tries was lowered to the complexity of rule applications. In an instance of the path consistency solver we observed a complexity that corresponds to the complexity of the best known algorithm for the problem. This solver consists of just one rule. Although we tried to produce examples that would exhibit the worst case behavior of the implementation, the empirical results are preliminary. At this stage of the research we cannot rule out with certainty that there are cases were the implementation actually shows the predicted worst-case complexity. Clearly more experiments are necessary.

Further work should take into account the effect of indexing and other optimizations in the complexity predictions. Another open question is which aspects in finding an appropriate ranking can be automated. We also would like to extend our approach to propagation rules. The difficulty is that for propagation rules, the ranking approach for derivation lengths does not apply. The approach of [McA99, GaMc01] also does not apply, since it does not deal with free variables at run-time and arbitrary built-in constraints.

Acknowledgements

Part of this work was performed while visiting the School of Computer Science and Software Engineering at Monash University, Melbourne, in March 2000 and while visiting the Department of Computer Science at the University of Pisa in October 2001.

References

[AAI00] Applied Artificial Intelligence, Special Issue on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), 2000.

[AbFr99] S. Abdennadher and T. Frühwirth, Operational Equivalence of CHR Programs and Constraints, 5th Intl Conf on Principles and Practice of Constraint Programming (CP'99), Springer LNCS 1894, 1999.

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

[Abd97] S. Abdennadher, Operational Semantics and Confluence of Constraint Propagation Rules, 3rd Intl Conf on Principles and Practice of Constraint Programming (CP'97), Springer LNCS 1330, 1997.

[Abd00] S. Abdennadher, A Language for Experimenting with Declarative Paradigms, Second Workshop on Rule-Based Constraint Reasoning and Programming, at the 6th Intl Conf on Principles and

Practice of Constraint Programming (CP'2000) Singapore, 2000.

[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), Elsevier, 1998.

[Fru00a] T. Frühwirth, Proving Termination of Constraint Solver Programs, in New Trends in Constraints, (K.R. Apt, A.C. Kakas, E. Monfroy and F. Rossi, Eds.), Springer LNAI 1865, 2000.

[Fru00b] T. Frühwirth, On the Number of Rule Applications in Constraint Programs, Declarative Programming - Selected Papers from AGP 2000, (A. Dovier, M. C. Meo, A. Omicini, Eds.), Electronic Notes in Theoretical Computer Science (ENTCS), Vol 48, Elsevier Science Publishers, June 2001.

[Fru01] T. Frühwirth, As Time Goes by: Complexity Analysis of Simplification Rules, Workshop on Quantitative Aspects of Programming Laguages (QAPL'01), at the Conf on Principles, Logics, and Implementations of high-level programming languages (PLI'01), Firenze, Italy, September 2001.

[GaMc01] H. Ganzinger and D. McAllester, A New Meta-Complexity Theorem for Bottum-up Logic Programs, (R. Gore, A. Leitsch and T. Nipkow, Eds.) First Intl Joint Conf on Automated Reasoning IJCAR 2001, Springer LNAI 2083, 2001.

[HoFr98] C. Holzbaur and T. Frühwirth, Constraint Handling Rules Reference Manual for Sicstus Prolog, TR-98-01, Österreichisches Forschungsinstitut für Artificial Intelligence, Vienna, Austria, July 1998.

[HoFr00] C. Holzbaur and T. Frühwirth, A Prolog Constraint Handling Rules Compiler and Runtime System, Applied Artificial Intelligence, Special Issue on Constraint Handling Rules (C. Holzbaur and T. Frühwirth, Eds.), Taylor & Francis, Vol 14(4), 2000.

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

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

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

[McA99] D. McAllester, On the Complexity Analysis of Static Analyses, (A. Cortesi and G. File, Eds.), 6th Intl. Static Analysis Symposium (SAS'99), Springer LNCS 1694, 1999.

[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.

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

[SaAb00] M. Saft and S. Abdennadher, WebCHR, Ludwig-Maximilians-Universität München, `www.pms.informatik.uni-muenchen.de/~webchr`, March 2000.

[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.