

Implementing and Analysing Union-Find in CHR

Tom Schrijvers, Thom Frühwirth

Report CW 389, July 2004



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Implementing and Analysing Union-Find in CHR

Tom Schrijvers, Thom Frühwirth

Report CW 389, July 2004

Department of Computer Science, K.U.Leuven

Abstract

CHR is a committed-choice rule-based language that was originally intended for writing constraint solvers. In this paper we show that it is also possible to write the classic union-find algorithm and variants in CHR. The programs neither compromise in declarativeness nor efficiency. Using CHR analysis techniques we study logical correctness, confluence and time complexity of our programs. We observe the essential destructive update of the algorithm. We can match the time complexity of the best known imperative implementations with the help of hashtable indexes. We illustrate the fact with experimental results.

Implementing and Analysing Union-Find in CHR

Tom Schrijvers* and Thom Frühwirth

¹ Department of Computer Science, K.U.Leuven, Belgium
www.cs.kuleuven.ac.be/~toms/

² Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. CHR is a committed-choice rule-based language that was originally intended for writing constraint solvers. In this paper we show that it is also possible to write the classic union-find algorithm and variants in CHR. The programs neither compromise in declarativeness nor efficiency. Using CHR analysis techniques we study logical correctness, confluence and time complexity of our programs. We observe the essential destructive update of the algorithm. We can match the time complexity of the best known imperative implementations with the help of hashtable indexes. We illustrate the fact with experimental results.

1 Introduction

When a new programming language is introduced, sooner or later the question arises whether classical algorithms can be implemented in an efficient and elegant way. For example, one often hears the argument that in Prolog some graph algorithms cannot be implemented with best known complexity because Prolog lacks destructive assignment that is needed for efficient update of the graph data structures. In particular, it is not clear if the union-find algorithm can be implemented with optimal complexity in pure (i.e. side-effect-free) Prolog [12].

In this programming pearl, we give a positive answer for the Constraint Handling Rule (CHR) programming language. We give an CHR implementation with the optimal worst case and amortized time complexity for the classical union-find algorithm with path compression for find and union-by-rank. This is particularly remarkable, since originally, CHR was intended for implementing constraint solvers only.

Relying on established analysis techniques for CHR, we show logical correctness, i.e. that the logical reading of the rules is a consequence of the theory of equivalence relations. The logical reading also shows that there is an inherent destructive update in the union-find algorithm. Next, the detailed confluence analysis helps to understand under which conditions the algorithm works as expected. It also shows in which way the results of the algorithm depend on the

* Research Assistant of the fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

order of its operations. Finally, the complexity analysis for CHR is extended to take indexing into account. In this way, the run-time-optimal implementation of the union-find algorithm in CHR can be shown.

Closest to this work is the presentation of a logical algorithm for the union-find problem in [12]. In a hypothetical bottom-up inference rule programming system with permanent deletions and rule priorities, a set of rules for union-find is given that is proven to run in $\mathcal{O}(M + N \log(N))$ worst-case time for a sequence of M operations on N elements. The direct efficient implementation of these inference rule system seems not feasible. For example, deleted atomic formulae have to be stored forever and new formulae have to be checked against them. It is also not clear if the rules given in [12] describe the standard union-find algorithm as can be found in text books such as [5]. The authors remark that giving a rule set with optimal amortized complexity is complicated. In contrast, we give an executable and efficient implementation that directly follows the pseudo-code presentations found in text books and that has also optimal amortized complexity. Moreover, we do not need to rely on rule priorities. We also analyse confluence and logical reading as well as logical correctness of our union-find program.

This paper is structured as follows. In the next Section, we review the classical union-find algorithm, several variations and related work. Constraint Handling Rules are briefly presented in Section 3. Then, in Section 4 we present the first basic implementation of the classical union-find algorithm in CHR. In Section 5 and Section 6 respectively the logical correctness and confluence properties of this CHR implementation are investigated. An improved version of the implementation, featuring path compression and union-by-rank, is presented next in Section 7. In Section 8, we argue that this implementation has the same time complexity as the best known imperative implementations. This claim is experimentally evaluated in Section 9. Finally, Section 10 concludes.

2 The Union-Find Algorithm

The classical union-find (also: disjoint set union) algorithm was introduced by Tarjan in the seventies [16]. The algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations on the sets:

- **make**(X): create a new set with the single element X .
- **find**(X): return the representative of the set in which X is contained.
- **union**(X, Y): join the two sets that contain X and Y , respectively (possibly destroying the old sets and changing the representative).

In the naive algorithm, these three operations are implemented as follows.

- **make**(X): generate a new tree with the only node X , i.e. X is the root.

- **find(X)**: follow the path from the node X to the root of the tree by repeatedly going to the parent node of the current node until the root is reached. Return the root as representative.
- **union(X,Y)**: find the representatives of X and Y, respectively. To join the two trees, it suffices to **link** them by making one root point to the other root.

The naive algorithm requires $\mathcal{O}(N)$ time per find (and union) in the worst case, where N is the number of elements (make operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve quasi-constant (i.e. almost constant) *amortized* running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root after a find. After **find(X)** returned the root of the tree, we make every node on the path from X to the root point directly to the root. A variant of path compression is also common. *Path-halving* means to directly point a node to its grandparent. It has the same time complexities and the advantage that the tree has to be only traversed once. We will see that in CHR one can do path compression also in one pass.

The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth. If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one. The variant *union-by-size* (meaning tree size, also called union-by-weight) has the same time complexities.

For each optimization alone and for using both of them together, the worst case time complexity for a single find or union operation is $\mathcal{O}(\log(N))$. For a sequence of M operations on N elements, the worst complexity is $\mathcal{O}(M + N\log(N))$. When both optimizations are used, the amortized complexity is quasi-linear, $\mathcal{O}(M + N\alpha(N))$, where $\alpha(N)$ is an inverse of the Ackermann function and is less than 5 for all practical N .

Current research in union-find algorithms deals with optimal trade-offs between worst-case and amortized complexities and between the complexity of the find and union operation under various computational models including parallelism. In particular, one can improve the worst-case complexity for union and find to $\mathcal{O}(\log(N)/\log\log(N))$ by using a more sophisticated compression for finds inside a union operation [4]. Another main strand of research considers dynamic versions of the algorithm, where edges in the tree can be deleted, where one can backtrack over union (i.e. undo the last union operation that has not been undone) or where any union can be undone. A classic survey on the topic is [11].

The union-find algorithm has applications in graph theory (e.g. efficient computation of spanning trees). We can also view the sets as equivalence classes with the union operation as equivalence. When the union-find algorithm is extended to deal with nested terms to perform congruence closure, the algorithm can be used for term unification in theorem provers and in Prolog.

The WAM [3], Prolog’s traditional abstract machine, uses the basic version of union-find for variable aliasing. While *variable shunting*, a limited form of

path compression, is used in some Prolog implementations [14], we do not know of any implementation of the optimized union-find that keeps track of ranks or other weights.

3 Constraint Handling Rules (CHR)

In this section we give an overview of syntax and semantics for constraint handling rules (CHR) [7, 10]. Readers familiar with CHR can skip this section.

CHR is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) into simpler ones until they are solved.

In CHR, one distinguishes two main kinds of rules: Simplification rules replace constraints by simpler constraints while preserving logical equivalence, e.g. $X \geq Y \wedge Y \geq X \Leftrightarrow X=Y$. Propagation rules add new constraints, which are logically redundant, but may cause further simplification, e.g. $X \geq Y \wedge Y \geq Z \Rightarrow X \geq Z$. The combination of propagation and multi-set transformation of logical formulae in a rule-based language that is concurrent, guarded and constraint-based make CHR a rather unique and powerful declarative programming language.

3.1 Syntax of CHR

We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols and CHR (user-defined) constraint symbols.

Built-in constraints are handled by a given, predefined constraint solver. We assume that these solvers are terminating and confluent. Built-in constraints include $=$, *true*, and *false*. The semantics of the built-in constraints is defined by a consistent first-order *constraint theory CT*. In particular, *CT* defines $=$ as the syntactic equality over finite terms.

CHR constraints are defined by a CHR program.

Definition 1. A *CHR program* is a finite set of rules. There are two kinds of rules:

A *simplification rule* is of the form

$$Name @ H \Leftrightarrow C \mid B.$$

A *propagation rule* is of the form

$$Name @ H \Rightarrow C \mid B,$$

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

A CHR constraint symbol is *defined* in a CHR program if it occurs in the head of a rule in the program.

Simpagation rules are a third, hybrid kind of CHR rule of the form

$$\text{Name} @ H \setminus H' \Rightarrow C \setminus B,$$

which abbreviate a simplification rule

$$\text{Name} @ H \wedge H' \Rightarrow C \setminus H \wedge B.$$

While we use simpagation rules in programs, the analyses will use their expanded form.

3.2 Declarative Semantics of CHR

The logical meaning of a simplification rule is a logical equivalence provided the guard holds. $\forall F$ denotes the universal closure of a formula F .

$$\forall(C \rightarrow (H \leftrightarrow \exists \bar{y} B)),$$

where \bar{y} are the variables that appear only in the body B .

The logical meaning of a propagation rule is an implication provided the guard holds

$$\forall(C \rightarrow (H \rightarrow \exists \bar{y} B)).$$

The logical meaning \mathcal{P} of a CHR program P is the conjunction of the logical meanings of its rules united with a consistent *constraint theory* CT that defines the built-in constraint symbols. We require CT to define the constraint symbol $=$ as syntactic equality.

3.3 Operational Semantics of CHR

The operational semantics of CHR is given by a transition system.

Let P be a CHR program. We define the transition relation \mapsto_P by introducing two computation steps (transitions), one for each kind of CHR rule (cf. Figure 1). Since the two computation steps (transitions) are structurally very similar, we first describe their common behavior and then explain the difference.

In Fig. 1, all meta-variables stand for (possibly empty) conjunctions of constraints. C and D stand for built-in constraints only, H and H' for CHR constraints only.

A *state* is simply a goal, i.e. a conjunction of built-in and CHR constraints. Conjunctions are considered as multi-sets of conjuncts (conjuncts can be permuted). We will usually partition a state into subconjunctions of specific kinds of constraints. For example, any state can be written as $(H' \wedge G \wedge D)$, where H' contains only CHR constraints, D only built-in constraints, and G arbitrary constraints. Each of the subconjunctions may be empty (equivalent to *true*).

A (fresh variant of a) rule is *applicable to a state* $(H' \wedge G \wedge D)$ if H' matches its head H and its guard C hold when the built-in constraints D of the state hold. A *fresh variant* of a rule is obtained by renaming its variables to fresh variables, denoted by the sequence \bar{x} .

Simplify

If $(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule in P with variables \bar{x}
and $CT \models \forall (D \rightarrow \exists \bar{x}(H=H' \wedge C))$
then $(H' \wedge G \wedge D) \mapsto_P^{\text{Simplify}} (G \wedge D \wedge B \wedge C \wedge H=H')$

Propagate

If $(H \Rightarrow C \mid B)$ is a fresh variant of a rule in P with variables \bar{x}
and $CT \models \forall (D \rightarrow \exists \bar{x}(H=H' \wedge C))$
then $(H' \wedge G \wedge D) \mapsto_P^{\text{Propagate}} (H' \wedge G \wedge D \wedge B \wedge C \wedge H=H')$

Fig. 1. Computation Steps of Constraint Handling Rules

Matching (one-sided unification) succeeds if H' is an instance of H , i.e. it is only allowed to instantiate (bind) variables of H but not variables of H' . Matching is logically expressed by equating H' and H but existentially quantifying all variables from the rule, that is \bar{x} . This equation $H'=H$ is shorthand for pairwise equating the arguments of the constraints in H' and H , provided their constraint symbols are equal.

If an *applicable rule is applied*, the equation $H=H'$, its guard C and its body B are added to the resulting state. Any one of the applicable rule can be applied (don't care non-determinism). A rule application cannot be undone - CHR is a committed-choice language without backtracking.

When a simplification rule is applied (transition **Simplify**), the matching CHR constraints H' are removed from the state. The **Propagate** transition is like the **Simplify** transition, except that it keeps the constraints H' in the resulting state. Trivial non-termination caused by applying a propagation rule again and again is avoided by applying it at most once to the same constraints [1].

A *computation* of a goal G in a program P is a sequence S_0, S_1, \dots of states with $S_i \mapsto_P S_{i+1}$ beginning with the initial state $S_0 = G$ and ending in a final state or diverging. \mapsto_P^* denotes the reflexive and transitive closure of \mapsto_P . A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent (unsatisfiable). When it is clear from the context, we will drop the reference to the program P .

Remark. Almost all current implementations of CHR follow a stricter, more refined operational semantics [6] in which the rules are tried in textual order only.

3.4 Well-Behavedness: Termination and Confluence

Typically, CHR programs for constraint solving are well-behaved, i.e. terminating and confluent. Confluence means that the result of a computation is independent from the order in which rules are applied to the constraints.

Definition 2. A CHR program is *well-behaved* if it is terminating and confluent.

A CHR program is called *terminating*, if there are no infinite computations.

For many existing CHR programs simple well-founded orderings are sufficient to prove termination [8]. Problems arise with non-trivial interactions between simplification and propagation rules.

Confluence guarantees that a CHR program will always compute the same result for a given query independent of which of the applicable rules are applied. This also implies that the order of constraints in a goal does not matter.

Definition 3. A CHR program is *confluent* if for all states S, S_1, S_2 : If $S \mapsto^* S_1$ and $S \mapsto^* S_2$ then there exist states T_1 and T_2 such that $S_1 \mapsto^* T_1$ and $S_2 \mapsto^* T_2$ and T_1 and T_2 are identical up to renaming of local variables and logical equivalence of built-in constraints.

The papers [1, 2] give a decidable, sufficient and necessary condition for confluence for terminating CHR programs. It is also shown that confluent CHR programs have a consistent logical reading.

For checking confluence, one takes copies (with fresh variables) of two rules (not necessarily different) from a terminating CHR program. The heads of the rules are *overlapped* by equating at least one head constraint from each rule. For each overlap, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a non-confluence. In any consistent state that contains the overlap of a non-joinable critical pair, the application of the two rules to the overlap will usually lead to different results.

Non-confluence can be repaired by *completion*. The automatic method is based on the idea of introducing rules between the two states in the critical pair. Sometimes, this is not possible, because the new rules cannot be made terminating, or we would have to introduce an infinite number of rules or ill-formed rules that have an empty head.

4 Implementing Union-Find in CHR

The following CHR program in concrete ASCII syntax implements the operations and data structures of the basic union-find algorithm without optimizations.

```

                                ufd_basic
make      @ make(A) <=> root(A).

union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B \ find(A,X) <=> find(B,X).
findRoot  @ root(A) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
link      @ link(A,B), root(A), root(B) <=> B ~> A, root(A).
```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the operations. `link/2` is an auxiliary relation for performing union. The constraints `root/2` and `~>/2` represent the tree data structure.

Remark. The use of the built-in constraint `=` in the rule `findRoot` is restricted to returning the element `A` in the parameter `X`, in particular no full unification is ever performed (that could rely on union-find itself).

Remark. The CHR constraint `root/1` can be dropped from the program if rules are applied in textual order only (as defined in the refined operational semantics of [6]).

Remark. The rule `link` can be interpreted as performing *abduction*. If the nodes `A` and `B` are not equivalent, introduce the minimal assumption `B ~> A` so that they are equivalent (i.e. performing union afterwards leads to application of rule `linkEq`).

As usual in union-find, we will allow the following queries (constraints):

- An *allowed query* consists of `make/1`, `union/2` and `find/2` constraints only. We call these the *external* operations (constraints). The other constraints (including those for the data structure) are generated and used internally by the program only.
- The elements we union are constants. A new constant must be introduced exactly once with `make/1` before being subject to `union/2` and `find/2`.
- The arguments of all constraints are constants, with exception of the second argument of `find/2` that must be a variable that will be bound to a constant, and the second argument of `root/2`, that must be an integer.

5 Logical Reading and Logical Correctness

The logical reading of our `ufd.basic` union-find CHR program is as follows:

<code>make</code>	$make(A) \Leftrightarrow root(A)$
<code>union</code>	$union(A, B) \Leftrightarrow \exists XY (find(A, X) \wedge find(B, Y) \wedge link(X, Y))$
<code>findNode</code>	$find(A, X) \wedge A \rightarrow B \Leftrightarrow find(B, X) \wedge A \rightarrow B$
<code>findRoot</code>	$root(A) \wedge find(A, X) \Leftrightarrow root(A) \wedge X = A$
<code>linkEq</code>	$link(A, A) \Leftrightarrow true$
<code>link</code>	$link(A, B) \wedge root(A) \wedge root(B) \Leftrightarrow B \rightarrow A \wedge root(A)$

From the logical reading of the rule `link` it follows that $B \rightarrow A \wedge root(A) \Rightarrow root(B)$, i.e. `root` holds for every node in the tree, not only for root nodes. Indeed, we cannot adequately model the update from a root node to a non-root node in first order logic, since first order logic is monotonic, formulas that hold cannot cease to hold. In other words, the link step is where the union-find algorithm is *non-logical* since it requires an update which is destructive in order to make the algorithm efficient.

By *logical correctness* we mean that the logical reading of the CHR rules is a consequence of an appropriate underlying theory. In the case of the union-find algorithm, the theory is that of equivalence relations. It consists of the axioms for reflexivity, symmetry and transitivity:

reflexivity	$A=A \Leftrightarrow true$
symmetry	$A=B \Leftrightarrow B=A$
transitivity	$A=B \wedge B=C \Rightarrow A=C$

To show logical correctness, we interpret all binary CHR constraints as equality $=$ and the unary constraints `make(A)` and `root(A)` as $A=A$, i.e. `true`:

<code>make</code>	$A=A \Leftrightarrow A=A$
<code>union</code>	$A=B \Leftrightarrow \exists XY(A=X \wedge B=Y \wedge X=Y)$
<code>findNode</code>	$A=X \wedge A=B \Leftrightarrow B=X \wedge A=B$
<code>findRoot</code>	$A=A \wedge A=X \Leftrightarrow A=A \wedge X=A$
<code>linkEq</code>	$A=A \Leftrightarrow true$
<code>link</code>	$A=B \wedge A=A \wedge B=B \Leftrightarrow B=A \wedge A=A$

It is easy to see that all formulas are logical consequences of the axioms for equality.

6 Confluence

We have analysed confluence of the union-find implementation with a small confluence checker written in Prolog and CHR. For the union-find implementation `ufd_basic`, we have found 8 non-joinable critical pairs. Two non-joinable critical pairs stem from overlapping the rules for `find`. Four non-joinable critical pairs stem from overlapping the rules for `link`. The remaining two critical pairs are overlaps between `find` and `link`.

We found one non-joinable critical pair that is unavoidable (and inherent in the union-find algorithm), three critical pairs that feature incompatible tree constraints (that cannot occur when computing allowed queries), and four critical pairs that feature pending link constraints (that cannot occur for allowed queries in the standard left-to-right execution order).

The Unavoidable Non-Joinable Critical Pair

The non-joinable critical pair between the rule `findRoot` and `link` exhibits that the relative order of `find` and `link` operations matters.

Overlap	<code>find(B,A),root(B),root(C),link(C,B)</code>
<code>findRoot</code>	<code>root(C),B~>C,A=B</code>
<code>link</code>	<code>root(C),B~>C,A=C</code>

It is not surprising that a `find` after a `link` operation has a different outcome if linking updated the root. As remarked in Section 5, this update is unavoidable and inherent in the union-find algorithm.

Incompatible Tree Constraints...

The two non-joinable critical pairs for `find` correspond to queries where a `find` operation is confronted with two tree constraints to which it could apply. Also the non-joinable critical pair involving the rule `linkEq` features incompatible tree constraints.

Overlap	$A \rightsquigarrow B, A \rightsquigarrow D, \text{find}(A, C)$
findNode	$A \rightsquigarrow B, A \rightsquigarrow D, \text{find}(B, C)$
findNode	$A \rightsquigarrow B, A \rightsquigarrow D, \text{find}(D, C)$
Overlap	$\text{root}(A), A \rightsquigarrow B, \text{find}(A, C)$
findNode	$\text{root}(A), A \rightsquigarrow B, \text{find}(B, C)$
findRoot	$\text{root}(A), A \rightsquigarrow B, A = C$
Overlap	$\text{root}(A), \text{root}(A), \text{link}(A, A)$
linkEq	$\text{root}(A), \text{root}(A)$
link	$\text{root}(A), A \rightsquigarrow A$

The conjunctions $(A \rightsquigarrow B, A \rightsquigarrow D)$, $(\text{root}(A), A \rightsquigarrow B)$, $(\text{root}(A), A \rightsquigarrow A)$ and $(\text{root}(A), \text{root}(A))$ that can be found in the overlaps (and non-joinable critical pairs) correspond to the cases that violate the definition of a tree: a node with two parents, a root with a parent, a root node that is its own parent, and a tree with two identical roots, respectively. Clearly, these four conjunctions should never occur during a run of the program.

...Cannot Occur

We show now that the four dangerous conjunctions indeed cannot occur as the result of running the program for an allowed query. We observe that the rule `make` is the only one that produces a `root`, and the rule `link` is the only one that produces a \rightsquigarrow . The rule `link` needs `root(A)` and `root(B)` to produce $A \rightsquigarrow B$, and it will absorb `root(A)`.

In order to produce one of the first three dangerous conjunctions, the link operation(s) need duplicate `root` constraints (as in the fourth conjunction) to start from. But only a query containing identical copies of `make` (e.g. `make(A), make(A)`) can produce the fourth dangerous conjunction. Since duplicate make operations are not an allowed query, we cannot produce any of the dangerous conjunctions (and non-joinable critical pairs) for allowed queries.

Adding Rules

We now investigate the question if we can add rules to repair the non-confluences. If this is the case we could relax the conditions on allowed queries (and execution order). All the 8 non-joinable critical pairs are in principle amenable to

completion, even though the automatically generated rules sometimes do not implement the intended behavior of the algorithm. We therefore also investigate which other rules can be added to make the program confluent.

For the unavoidable non-joinable critical pair completion would add a rule that in effect equates B and C, a behavior that is not intended and that will lead to failure in most cases:

```
findRoot - link      root(C), B~>C <=> A=B | A=C.
```

```
findNode - findNode A~>B, A~>D \ find(B,C) <=> B>D | find(D,C).
```

```
findNode - findRoot root(A), A~>B \ find(B,C) <=> A=C.
```

```
linkEq   - link      root(A) \ A~>A <=> root(A).
```

The rule generated from the `findNode`-`findNode` critical pair needs a guard to ensure termination. This rule makes the `find` operation behave on $A \sim B, A \sim D$ as if it was $A \sim B, B \sim D$. (For $B=D$ the two states in the critical pair become identical.) The rule added for `findNode`-`findRoot` means that $A \sim B$ will be ignored since A is the root. The rule for `linkEq`-`link` replaces $A \sim A$ by `root(A)` when there is already a `root(A)`. (The non-joinable critical pair between `linkEq` - `link` could be avoided altogether by making sure in the guard of the `link` rule that the arguments of `link/2` are different.)

The logical reading of the last rule implies $\text{root}(A) \Rightarrow A \sim A$. This reminds us that in most presentations as well as implementations of the union-find algorithm [5], roots of a tree are represented by a pointer variable that points to itself, i.e. $A \sim A$. Therefore we may assert $\text{root}(A) \Leftrightarrow A \sim A$. Indeed, in our program `ufd_basic` we could uniformly replace `root(A)` by $A \sim A$ for any A, and guard the `findNode` rule against infinite recursion by requiring that $A \neq B$ (there are even CHR implementations that will terminate without the guard).

The rules generated automatically by completion are not really satisfactory. With rules introduced by hand one can directly address the incompatible tree constraints. Once we have $\text{root}(A) \Leftrightarrow A \sim A$, we can neutralize the three critical pairs by the following rules that involve the dangerous conjunctions of incompatible tree constraints:

```
A~>B \ A~>C <=> A=C.
root(A) \ A~>B <=> A=B.
root(A) \ root(A) <=> true.
```

The first rule expresses the functional dependency in the arguments of \sim . (We may also replace the built-in equality = by `union`.)

While logically sound, the rules that equate nodes will usually lead to failure of the computation. Also with the introduction of these additional rules, the complexity of the overall algorithm may change.

Pending Links...

The remaining four non-joinable critical pairs stem from overlapping the rule for `link` with itself. They correspond to queries where two `link` operations have at

least one node in common such that when one link is performed, at least one node in the other link operation is not a root anymore.

Overlap	$\text{root}(A), \text{root}(B), \text{link}(B, A), \text{link}(A, B)$
link	$\text{root}(B), A \rightsquigarrow B, \text{link}(A, B)$
link	$\text{root}(A), \text{link}(B, A), B \rightsquigarrow A$
Overlap	$\text{root}(A), \text{root}(B), \text{root}(C), \text{link}(B, A), \text{link}(C, B)$
link	$\text{root}(C), A \rightsquigarrow B, B \rightsquigarrow C$
link	$\text{root}(A), \text{root}(C), \text{link}(B, A), B \rightsquigarrow C$
Overlap	$\text{root}(A), \text{root}(B), \text{root}(C), \text{link}(B, A), \text{link}(A, C)$
link	$\text{root}(B), \text{root}(C), A \rightsquigarrow B, \text{link}(A, C)$
link	$\text{root}(B), C \rightsquigarrow A, A \rightsquigarrow B$
Overlap	$\text{root}(A), \text{root}(B), \text{root}(C), \text{link}(B, A), \text{link}(C, A)$
link	$\text{root}(B), \text{root}(C), A \rightsquigarrow B, \text{link}(C, A)$
link	$\text{root}(B), \text{root}(C), \text{link}(B, A), A \rightsquigarrow C$

...Cannot Occur

When we analyse these non-joinable critical pairs we see that the two conjunctions $(A \rightsquigarrow C, \text{link}(A, B))$ and $(A \rightsquigarrow C, \text{link}(B, A))$ are dangerous.

Once again, we argue now that the critical pairs can never arise in practice in an allowed query. `link` is an internal operation, it can only be the result of a `union`, which is an external operation. In the union, the link constraint gets its arguments from `find`. In the standard left-to-right execution order of most sequential CHR implementations [6], first the two find constraints will be executed and when they have finished, the link constraint will be processed. In addition, no other operations will be performed inbetween these operations. Then the results from the find constraints will still be roots when the link constraint receives them. Note that such an execution order is always possible, provided `make` has been performed for the nodes that are subject to union (as is required for allowed queries).

Otherwise (when `make` is called too late or when execution is interleaved or in parallel) we may end up with pending link constraints. For example, the left-to-right execution of the query `make(A), union(A, B), union(A, B), make(B)` results in $\text{root}(A), \text{link}(A, B), B \rightsquigarrow A$, because the second `make` constraint comes too late.

Adding Rules

Note that the second and third critical pair are identical up to variable renaming. From these two critical pairs completion can derive a rule that reduces the remaining `link`:

$$\text{root}(B), A \rightsquigarrow B \ \backslash \ \text{root}(C), \text{link}(A, C) \ \Leftrightarrow \ C \rightsquigarrow A.$$

But for the other two critical pairs, completion cannot get rid of the pending link constraint, for example consider:

$$\text{root}(B), \text{root}(C) \setminus A \rightsquigarrow B, \text{link}(C,A) \Leftrightarrow B > C \mid A \rightsquigarrow C, \text{link}(B,A).$$

However, we can add rules for the `link` to neutralize the dangerous conjunctions:

$$\begin{aligned} A \rightsquigarrow C \setminus \text{link}(A,B) &\Leftrightarrow \text{link}(C,B). \\ A \rightsquigarrow C \setminus \text{link}(B,A) &\Leftrightarrow \text{link}(B,C). \end{aligned}$$

To the same effect, we may rewrite `link` into `union`:

$$\begin{aligned} A \rightsquigarrow C \setminus \text{link}(A,B) &\Leftrightarrow \text{union}(A,B). \\ A \rightsquigarrow C \setminus \text{link}(B,A) &\Leftrightarrow \text{union}(A,B). \end{aligned}$$

That means that we are restarting `union` when we have to link a non-root.

Note that when adding either of the two pairs of rules, we will still get non-joinable critical pairs, but they will only differ in structure of the tree, i.e. the critical pairs are joinable modulo equivalence of the set of nodes of the trees. Here the rules are potentially useful in cases where we do not have a strict left-to-right-execution order, e.g. in parallel implementations.

7 Optimized Union-Find

The following CHR program implements the optimized classical Union-Find Algorithm with path compression for find and union-by-rank [16].

```

                                ufd_rank
make      @ make(A) <=> root(A,0).

union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B, find(A,X) <=> find(B,X), A ~> X.
findRoot  @ root(A,_) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
linkLeft  @ link(A,B), root(A,N), root(B,M) <=> N>=M |
           B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight @ link(B,A), root(A,N), root(B,M) <=> N>=M |
           B ~> A, N1 is max(N,M+1), root(A,N1).
```

When compared to the basic version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. The rule `findNode` has been extended for path compression already during the first pass along the path to the root of the tree. This is achieved by the help of the logical variable `X` that serves as a place holder for the result of the find operation. The

`link` rule has been split into two rules `linkLeft` and `linkRight` to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is chosen (both rules are applicable) and the rank is incremented by one.

Remark. Path compression (cf. rule `findNode`) can be interpreted as *memoization* or *tabling* of all the (intermediate) results of the recursive find operation, where the memoized `find(A,X)` is stored as $A \rightsquigarrow X$.

The results for logical reading and logical correctness of the optimized union-find are analogous to the ones for `ufd_basic`.

Confluence Revisited

The non-joinable critical pairs (CPs) are in principle analogous to the ones discussed for `ufd_basic` in Section 6, but their numbers significantly increases due to the optimizations of path compression and union-by-rank that complicate the rules for the find and link operations.

Our confluence checker found 73 non-joinable critical pairs. The number of critical pairs is dominated by those of the link rules. Not surprisingly, each critical pair involving `linkLeft` has a corresponding analogous critical pair involving `linkRight`.

Rule1	Rule2	Number of CPs
<code>findRoot</code>	<code>linkLeft</code>	1
<code>findRoot</code>	<code>linkRight</code>	1
<code>findNode</code>	<code>findNode</code>	2
<code>findNode</code>	<code>findRoot</code>	1
<code>linkEq</code>	<code>linkLeft</code>	1
<code>linkEq</code>	<code>linkRight</code>	1
<code>linkLeft</code>	<code>linkLeft</code>	20
<code>linkLeft</code>	<code>linkRight</code>	26
<code>linkRight</code>	<code>linkRight</code>	20

We discuss the interesting critical pairs (as it turns out, those involving `find` operations) in detail, and the others in bulk.

The CPs between `findRoot` and a link rule are the unavoidable critical pairs as in `ufd_basic`. These show the expected behavior that the result of `find` will differ if its executed before or after a link operation, for example:

Overlap	<code>find(B,A), root(B,C), link(E,B), root(E,D), D>=C</code>
<code>findRoot</code>	$A=B, D>=C, N \text{ is } \max(D, C+1), \text{root}(E, N), B \rightsquigarrow E$
<code>linkLeft</code>	$A=E, D>=C, N \text{ is } \max(D, C+1), \text{root}(E, N), B \rightsquigarrow E$

Two `findNode` rule applications on the same node will interact, because one will compress, and then the other cannot proceed until the first find operation has finished:

Overlap	<code>find(B,A),B~>C,find(B,D)</code>
<code>findNode</code>	<code>find(A,D),find(C,A),B~>D</code>
<code>findNode</code>	<code>find(D,A),find(C,D),B~>A</code>

We see that A and D are interchanged in the states of the critical pair. In the first state, since the result of `find(C,A)` is A, the `find(A,D)` can eventually only reduce to A=D. Analogously for the second state. But under A=D the two states of the critical pair are identical. We could add a rule `find(C,A) \ find(A,D) <=> A=D` to repair this critical state.

The other two critical pairs involving a `findNode` rule correspond to impossible queries `B~>C,B~>D` and `root(B,N),B~>C` as discussed for the confluence of `ufd_basic`. However, because of path compression, the rules suggested to be added in that section do not work any more.

All critical pairs between link rules only, except those for `linkEq`, consist of pairs of states that have the same constraints and variables, but that differ in the tree that is represented. Just as in the case of `ufd_basic` the problem of pending links occurs without a left-to-right execution order.

However, a number of these critical pairs is spurious, because the confluence checker does not know about functional dependencies in the constraints: The first arguments of `root/2` and `find/2` determine their respective second arguments, and the second argument of `is/2` determines its first argument.

Overall, 9 of the 20 CPs between the same link rules involve multiple occurrences of `root` constraints for the same node. This also holds for 13 of the 28 CPs between different link rules (including the 2 CPs for `linkEq`). So these CPs can be ignored, because duplicate root constraints cannot occur in computations for allowed queries.

The CPs between `linkEq` and other link rules could be avoided as in `ufd_basic` by introducing a proper guard. The CPs between the `linkLeft` and the `linkRight` rule could be avoided by strengthening the guard on one of the rules so that the ranks must be different.

8 Worst-Case Time Complexity

We will build on the automatic complexity analysis results for CHR [9]. The complexity measure is basically the derivation length times the sum of the costs of rule trials and a rule application. The dominating cost factor is the computation of the cross product of all constraints in a goal when trying to apply a rule. However, often and also here, this cost can be made constant by relying on execution orders (scheduling) of constraints and indexing techniques.

More precisely, in [9] the following bound on the worst case time complexity for a query with n atomic CHR constraints has been derived for running programs of simplification rules in any CHR implementation (independent of any optimizations).

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

where

- c is the number of constraint in the query Q
- $D(= |Q|)$ is the worst case derivation length of the query Q
- i ranges over the rules in the CHR program
- n_i is the number of heads in rule i
- O_{H_i} is the complexity of the head matchings for rule i
- O_{G_i} is the complexity of the guard for rule i
- O_{C_i} is the complexity of adding the rhs built-in constraints for rule i
- O_{B_i} is the complexity of removing the lhs CHR constraints and of adding the rhs CHR constraints for rule i

These assumptions will underly our complexity analysis for union-find:

- The query consists of N `make/1` constraints for different elements (nodes), followed by M `union/2` and `find/2` constraints, arbitrarily ordered. Hence, $c = \mathcal{O}(N + M)$.
- All of O_{H_i} , O_{G_i} , O_{C_i} and O_{B_i} are constants. This means that rule applications can be done in constant time, while rule trials may cost more.
- Simpagation rules are rewritten into the corresponding simplification rules.

Worst-Case Derivation Length

We derive the derivation length from a termination order that uses the following mapping of constraints and constants onto natural numbers:

- $|C \wedge D| = |C| + |D|$.
- $|A=B| = |\text{root}(A, N)| = |A \rightsquigarrow B| = 0$.
- $|\text{make}(A)| = |\text{link}(A, B)| = 1$.
- $|\text{find}(A, B)| = |A| + 1$.
- $|\text{union}(A, B)| = |A| + |B| + 4$.
- $|A|$ is the path length (distance) from the node A to the root of the tree.

That is, termination and the order of the derivation length of a query depends only on the length of the recursion of `find/2`. The upper bounds for the depth of the tree - and thus the path length - are given by the number N of elements and also by the rank, the second argument of `root/2`. The order of the derivation length of a single find, i.e. $|A|$ of its first argument, can be bound by $|A| \leq \log_2(N)$. This can be easily shown by induction on the rank.

Putting it all together, the derivation length of a query with N `make/1` constraints and M `union/2` and `find/2` constraints is $D = \mathcal{O}(N + M \log(N)) = c + D$.

Taking into account that $\max_i(n_i) = 3$, we get the following complexity bound:

$$\mathcal{O}((N + M \log(N))^4)$$

Worst-Case Time Complexity with Indexing

We can reduce this very crude upper bound by taking the actual operational semantics of sequential implementations of CHR [6] and indexing into account. In that case we can reduce the cost of rule trials to constant time so that the derivation length directly gives the complexity.

Theorem. The worst case complexity for an allowed query with N different **make** operations (i.e. N nodes) and M **union** and **find** operations for the CHR program **ufd** is:

$$\mathcal{O}(N + M \log(N))$$

Proof Sketch. We assume a sequential CHR implementation with left-to-right execution order and indexing. The proof is based on the following observations:

1. Since a given query is executed from left to right, its external operations **make/1**, **union/2** and **find/2**, as well as the internal operation **link/2** can always immediately reduce due to a rule application. This is obvious for **make/1** and **union/2**. For **find/2** observe that a root node with a rank must have been introduced for its first argument by **make/1**. By the **link/2** rules, the rank of the involved nodes has either been left as is, increased or been removed for an edge. For **link/2** observe that at least one of its three rules must apply, because it is applied to the results of the two **find** operations.
2. The introduction of **root/2** and **~>/2** constraints in the body of rules cannot trigger any immediate rule applications: If a given conjunction of constraints is executed from left to right, there will be no pending operations (see observation 1).
3. The compiler can automatically put an index on the common variables of different constraints in the multi-headed rules for **find/2** and **link/2**, i.e. on the first arguments of **find/2** and **link/2**, as well as of **~>** and **root**, and in addition on the second argument of **link/2**. Since these rules will be triggered by the operations (observation 1), it is sufficient to place an index on the first arguments of **~>** and **root**. Due to constant-time indexing we can find the partner constraints in constant time, since allowed queries produce proper tree constraints where a node occurs exactly once as the first argument of either **~>/2** or **root/2**.

Due to indexing (observation 3) and because any operation can trigger rules (observation 2), the rules behave like single-headed rules complexity-wise. Due to observation 1 and 2 we know that the next constraint to process is either immediately reduced by a rule application (for operations) or no rule is applicable (for tree constraints), and both computations can be done in constant time (observation 3).

Hence, in the complexity formula, n_i reduces to 0, and the actual complexity is just $\mathcal{O}(N + M \log(N))$.

Optimal Time Complexities

This complexity is worse than $\mathcal{O}(M+N\log(N))$ for $M > N$. The automatic complexity analysis of [9] that we have used does not seem to cover the subtle effects of path compression. Consider a goal `find(A,X)`. For each (but the last) application of the compressing `findNode` rule, we remove one current intermediate node `B` from the path from `A` to the root (that will be returned as the value of `X`). The cost of a find operation is thus linear in the number of removed nodes. These removed nodes can never be reintroduced, but the link operations can extend the path in front by changing the root. Due to union-by-rank, the longest possible paths cannot have more than $\log(N)$ nodes. A find operation for node `A` will remove some of these nodes on the path, and subsequent find operations for the same node will not need to traverse them again. Hence the overall complexity of m find operations for the same node is bounded by $\mathcal{O}(m + \log(N))$. For M arbitrary find operations involving N nodes, we will have a cost of $\mathcal{O}(M + N\log(N))$. Since the make and link operations take constant time each, this is also the worst-case time complexity of the union-find algorithm.

We also claim that the algorithm has optimal amortized complexity of $\mathcal{O}(M + N\alpha(N))$ (which is rather hard to show [5, 16]), because it faithfully and straightforwardly implements the pseudo-code of the union-find algorithm that can be found in text books, e.g. [5]. The previous proof showed that there is no overhead in implementing the operations of union-find with CHR rules, since they can be applied in constant time. The claim is supported by the experimental evaluation discussed in the next section.

9 Experimental Evaluation

To experimentally validate the derived complexity derived above, we have run the CHR program in SWI-Prolog [17] using the K.U.Leuven CHR system [15]. To make sure that the proper indexes are used that allow $\mathcal{O}(1)$ lookup, mode declarations have been added:

ufd_rank mode declarations

```
option(mode,make(+)).
option(mode,rank(+,+)).
option(mode,union(+,+)).
option(mode,find(+,?)).
option(mode,link(+,+)).
option(mode,'~>'(+,+)).
```

The meaning of this mode information is as follows: a `+` means that the corresponding argument is ground at all times and a `?` means that nothing is known.

For indexes on ground arguments, the compiler can use hashtables as constraint stores: a hash function for ground terms is defined in SWI-Prolog which yields an integer hash value in $\mathcal{O}(1)$ time for atoms. This hash value is used for

$\mathcal{O}(1)$ lookup in a hash table. In addition the hashtable guarantees $\mathcal{O}(1)$ insertion and deletion. The latter holds provided that the term appears just once as a key to the index, but that is the case in the `ufd_rank` program. These complexity bounds on insertion and deletion ensure that $O_{B_i} = \mathcal{O}(1)$.

In contrast, the first and de facto standard CHR system, available in SICStus [13], does not provide the necessary constant time operations. While it does have constant lookup time for all constraint instances of a particular constraint that contain a particular variable, it does not distinguish between argument positions. Hence, the lookup of `rank(X,R)` can be done in constant time given `X`, but the lookup of `X ~> Y` is proportional to the number of `~>` constraints `X` appears in. If `X` is a node with K children, then it will be $\mathcal{O}(K)$. Moreover, while the insertion of a constraint instance is $\mathcal{O}(1)$, deletion is $\mathcal{O}(I)$, where I is the total number of instances of the constraint.

The queries we use in our experimental evaluation consist of N calls to `make/1`, to create N different elements, followed by N calls to `union/2` and N calls to `find/2`. The input arguments of the latter two are chosen at random among the elements. Figure 2 shows a plot of N versus the runtime (in milliseconds). The expected near-linear behavior is clearly visible.

However, even the SICStus CHR system shows near-linear behavior for a random set of `union` operations. So let us consider instead a contrived set of `union` operations: disjoint trees of elements are unioned pairwise until all elements are part of the same tree. Figure 3 shows the runtime results for SICStus and SWI-Prolog. It is clear from the figure that SICStus does not show the optimal quasi-linear behavior anymore which is still observed in SWI-Prolog.

This illustrates that it is vital for efficiency to use a CHR system with the proper constraint store data structures. To the best of our knowledge, the K.U.Leuven CHR system is currently the only system that provides hashtable-based indexing constraint stores.

10 Conclusion

We have shown in this paper that it is possible to implement the classical union-find algorithm concisely and efficiently in constraint handling rules (CHR). The implementation is easily extended with optimizations like path-compression and union-by-rank. In addition, we have used and adapted established reasoning techniques for CHR to show the logical correctness, confluence and time complexity properties of our implementations. The logical reading and the confluence check showed the essential destructive update of the algorithm when trees are linked. Non-confluence can be caused by incompatible tree constraints (that cannot occur when computing with allowed queries), and due to competing link operations (that cannot occur with allowed queries in the standard left-to-right execution order). Last but not least, it turned out that the declarative nature of CHR is no compromise for time complexity.

At <http://www.cs.kuleuven.ac.be/~toms/Research/CHR/UnionFind/> all presented programs as well as related material are available for download. The

programs can be run with the proper time complexity in the latest release of SWI-Prolog.

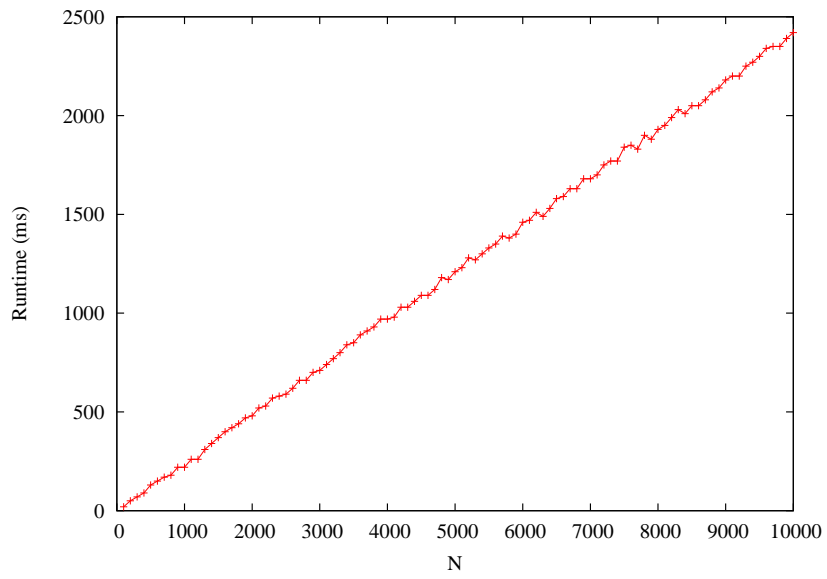
In future work we intend to investigate implementations for other variants of the union-find algorithm. For a parallel version of the union-find algorithm parallel operational semantics of CHR have to be investigated (confluence may be helpful here). A dynamic version of the algorithm, e.g. where unions can be undone, would presumably benefit from dynamic CHR constraints as defined in [18].

Acknowledgements. We would like to thank the participants of the first workshop on CHR for raising our interest in the subject. Marc Meister and the students of the constraint programming course at the University of Ulm in 2004 helped by implementing and discussing their versions of the union-find algorithm.

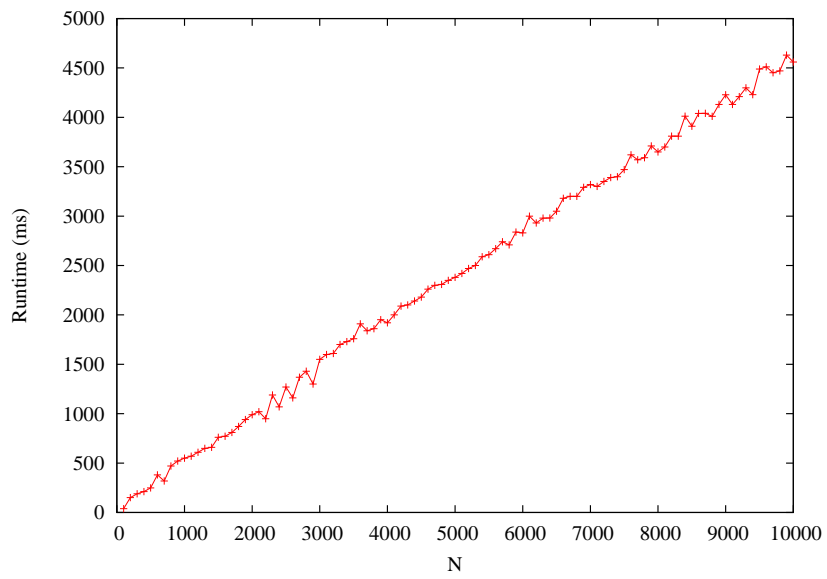
References

1. S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.
2. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints Journal*, 4(2), 1999.
3. H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
4. S. Alstrup, A. M. Ben-Amram, and T. Rauhe. Worst-case and amortised optimality in union-find. In *STOCS*, 1999.
5. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
6. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The refined operational semantics of constraint handling rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, 2004.
7. T. Frühwirth. Theory and practice of constraint handling rules, special issue on constraint logic programming. *Journal of Logic Programming*, pages 95–138, October 1998.
8. T. Frühwirth. As time goes by: Automatic complexity analysis of simplification rules. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, 2002.
9. T. Frühwirth. As time goes by II: More automatic complexity analysis of concurrent rule programs. In A. D. Pierro and H. Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2002.
10. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
11. Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
12. H. Ganzinger and D. McAllester. A new meta-complexity theorem for bottom-up logic programs. In *International Joint Conference on Automated Reasoning*, LNCS 2083, pages 514–528. Springer, 2001.

13. Intelligent Systems Laboratory. *SICStus Prolog User's Manual*. PO Box 1263, SE-164 29 Kista, Sweden, October 2003.
14. D. Sahlin and M. Carlsson. Variable Shunting for the WAM. Technical Report SICS/R-91/9107, SICS, 1991.
15. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In T. Frühwirth and M. Meister, editors, *First Workshop on Constraint Handling Rules: Selected Contributions*, number 2004-01, 2004. ISSN 0939-5091.
16. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.
17. J. Wielemaker. SWI-Prolog. <http://www.swi-prolog.org/>.
18. A. Wolf. Adaptive constraint handling with chr in java. In *7th International Conference on Principles and Practice of Constraint Programming (CP 2001)*, LNCS 2239. Springer, 2001.

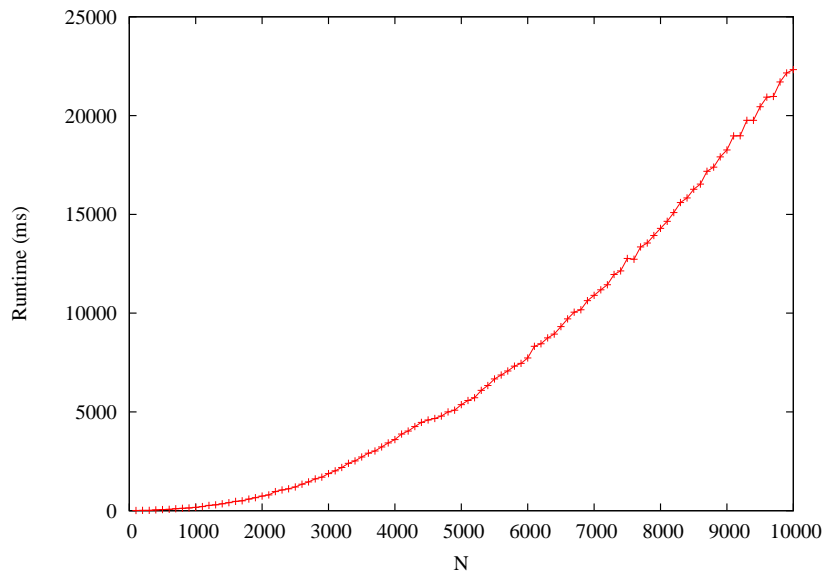


(a) SICStus

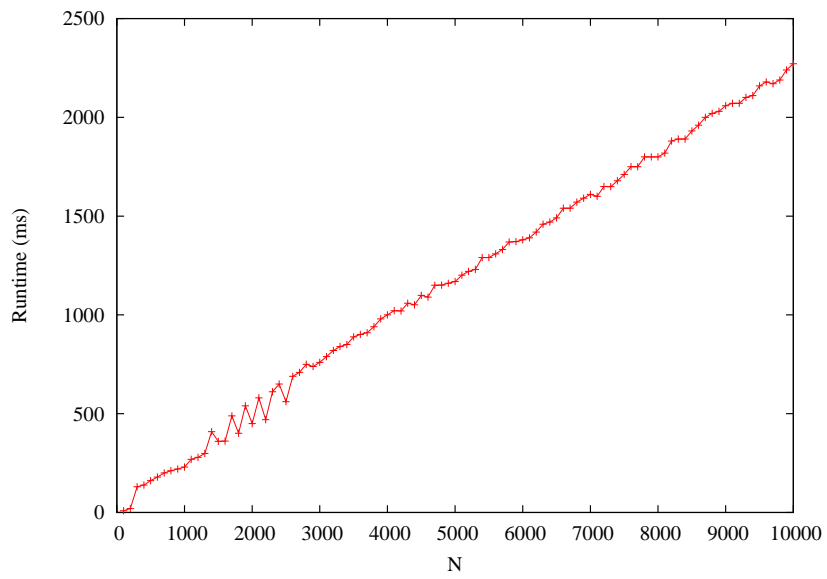


(b) SWI-Prolog

Fig. 2. Observation of near-linear behavior for random unions.



(a) SICStus



(b) SWI-Prolog

Fig. 3. Observation of behavior for contrived unions.