Deriving Quasi-Linear-Time Algorithms from Union-Find in CHR
Extended Abstract

Thom Frühwirth

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

Abstract. The union-find algorithm can be seen as solving simple equations between variables or constants. With a few lines of code change, we generalise its implementation in CHR from equality to arbitrary binary relations. By choosing the appropriate relations, we can derive fast algorithms for solving certain propositional logic (SAT) problems as well as certain polynomial equations in two variables. While linear-time algorithms are known to check satisfiability and to exhibit certain solutions of these problems, our algorithms are simple instances of the generic algorithm and have additional properties that make them suitable for incorporation into constraint solvers: From classical union-find, they inherit simplicity and quasi-linear time and space. By nature of CHR, they are anytime and online algorithms. They can be parallelised. They solve and simplify the constraints in the problem, and can test them for entailment, even when the constraints arrive incrementally, one after the other. We show that instances where relations are bijective functions yield precise and correct algorithm instances of our generalised union-find.

1 Introduction

Constraint Handling Rules (CHR) [6,8] is a logical constraint-based concurrent committed-choice programming language consisting of guarded rules that rewrite multisets of atomic formulas.

It was shown recently that the classical optimal *union-find* algorithm [15] is implementable in CHR with best-known *quasi-linear time* complexity [12,13]. This result is not accidental, since the paper [14] shows that a subset of the CHR language can simulate Turing and RAM machines in polynomial time, thus establishing that CHR is Turing-complete and, more importantly, that every algorithm can be implemented in CHR with best known time and space complexity, something that is not known to be possible in other pure declarative programming languages.

The union-find algorithm maintains disjoint sets under the operation of union. By definition of set operations, a union operator working on representatives of sets is an equivalence relation, i.e. we can view sets as equivalence classes. Especially iff the elements of the set are variables or constants, union can be seen as equating those elements and giving an efficient way of finding out if two elements are equivalent (i.e., in the same set).

This extended abstract investigates the question if the union-find algorithm written in CHR can be generalised so that other relations than simple equations between two variables are possible without compromising efficiency.

Overview of the Paper. This extended abstract discusses the following topics in the next sections.

- Quasi-Linear Time Union-Find Algorithm
- Generalised Union-Find in CHR
- Example Instances: Boolean and Polynomial Equations in Two Variables
- Time and Space Complexity and Correctness of the Generalisation
- Conclusions and Future Work

We assume some familiarity with CHR [6,8] in this extended abstract.

2 The Union-Find Algorithm

In this section we follow the exposition of [12]. The classical union-find (also referred to as disjoint-set-union) algorithm was introduced by Tarjan in the seventies [15]. A classic survey on the topic is [9]. 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 set 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 changing the representative).

A new element must be introduced exactly once with `make` before being subject to `union` and `find` operations. To find out if two elements are in the same set already, i.e. to check entailment, one finds their representatives and checks them for equality, i.e. checks `find(X)=find(Y)`.

2.1 Implementing Union-Find in CHR

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

In the CHR implementation, the constraints make/1, union/2, find/2 and link/2 define the operations (functions are written in relational form), so we call them *operation constraints*. The constraints root/1 and ->/2 (using infix notation) represent the tree data structure and we call them *data constraints*. We use the infix notation ->/2 to evoke the image of a directed arc, since it is often helpful for the understanding of the algorithm to imagine the tree as directed graph.

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

findNode @ X -> Y, find(X,R) <=> X -> Y, find(Y,R).
findRoot @ root(X), find(X,R) <=> root(X), R=X.

linkEq   @ link(X,X) <=> true.
link     @ link(X,Y), root(X), root(Y) <=> X -> Y, root(Y).
```

2.2 Optimised Union-Find

The basic 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 optimisations that keep the tree shallow and balanced, one can achieve logarithmic worst-case and quasi-constant (i.e. almost constant) amortised running time per operation.

The first optimisation 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.

The second optimisation 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 (tree height). If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one. With this optimisation, the height of the tree can be logarithmically bound.

The following CHR program implements the optimised classical union-find algorithm with path compression for find and union-by-rank [15].

```
make     @ make(X) <=> root(X,0).
union    @ union(X,Y) <=> find(X,A), find(Y,B), link(A,B).

findNode @ X -> Y, find(X,R) <=> find(Y,R), X -> R.
findRoot @ root(X,N), find(X,R) <=> root(X,N), R=X.

linkEq   @ link(X,X) <=> true.
linkLeft @ link(X,Y), root(X,RX), root(Y,RY) <=> RX>=RY |
```

```
              Y -> X, root(X,max(RX,RY+1)).
linkRight@ link(X,Y), root(Y,RY), root(X,RX) <=> RY>=RX |
              X -> Y, root(Y,max(RY,RX+1)).
```

When compared to the naive version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. The `union/2` operation constraint is implemented exactly as for the naive algorithm. The rule `findNode` has been extended for path compression. By the help of the variable `R` that serves as a place holder for the result of the find operation, path compression is already achieved during the first pass, i.e. during the find operation. The `link` rule has been split into two rules, `linkLeft` and `linkRight`, to reflect the optimisation 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 updated (both rules are applicable) and the rank is incremented by one.

3 Generalised Union-Find

The idea of generalising union find is to replace equations between variables by arbitrary binary relations. The operation `union` now asserts a given relation between its two variables, `find` finds the relation between a given variable and the root of the tree in which it occurs. The operation `link` includes the relation as well so that it is stored in the tree data constraint, i.e. the arcs in the tree are labelled by relations now. In the CHR implementation, the operation constraints `union`, `find`, `link` and the data constraint `->` get an additional argument to hold the relation.

We also need some standard *operations on relations* from relational algebra that are implemented by constraints as follows (where we use relational notation and *id* is the identity function, i.e. equality):

- `compose`(r_1, r_2, r_3) iff $r_1 \circ r_2 = r_3$
- `invert`(r_1, r_2) iff $r_1 = r_2^{-1}$
- `equal`(r_1) iff $r_1 = id$

The following code extends the CHR implementation of optimal union-find by additional arguments and by additional constraints on them. These additions are emphasised for clarity. Our implementation in Sicstus Prolog CHR is available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/ufe.pl`.

```
make     @ make(X) <=> root(X,0).
union    @ union(X,XY,Y) <=> find(X,XA,A), find(Y,YB,B),
                         combine(XA,YB,XY,AB), link(A,AB,B).

findNode @ X-XY->Y, find(X,XR,R) <=> find(Y,YR,R),
                         compose(XY,YR,XR), X-XR->R.
findRoot @ root(X,N), find(X,XR,R) <=> root(X,N), equal(XR), X=R.
```

```
linkEq   @ link(X,XX,X) <=> equal(XX).
linkLeft @ link(X,XY,Y), root(X,RX), root(Y,RY) <=> RX>=RY |
           invert(XY,YX), Y-YX->X, root(X,max(RX,RY+1)).
linkRight@ link(X,XY,Y), root(Y,RY), root(X,RX) <=> RY>=RX |
                         X-XY->Y, root(Y,max(RY,RX+1)).
```

The operation constraint `union(X,XY,Y)` now means that we enforce relation `XY` between `X` and `Y`. The operation `find` still returns the root for a given node, but also the relation that holds between the node and the root. From these three relations, `combine` derives the relation `AB` that must hold between the roots that are to be linked.

The `combine` operation can be defined in CHR as follows:

```
combine(XA,YB,XY,AB) <=>                X -- XA* -- A
    compose(XY,YB,XB),                  |           |
    invert(XA,AX),                      XY         AB?
    compose(AX,XB,AB).                  |           |
                                        Y -- YB* -- B
```

The crude graphics on the right of the CHR code shows the relations between the four relations that are arguments of `combine`. It is a so-called *commutative diagram*. The question mark after `AB` reminds us that this relation is the one that `combine` computes from the other three. The starred relations `*` remind us that `find` computes these relations by repeated composition of the relations on the path of the tree. Overall, given the relations between `X` and `Y`, `X` and `A`, `Y` and `B`, `combine` computes the relation between `A` and `B` that will replace the relation `XY` in the tree representation.

4 Instance of Boolean Equations

With our generalised union-find algorithm, we can solve inequations between Boolean variables (propositions), i.e. certain 2-SAT problems. This instance features a finite domain and a finite number of relations. In the CHR implementation, the relations are `eq` for $=$ and `ne` for $\neq$, and the truth values are `0` for false and `1` for true. The operations on relations can be defined by the following facts:

```
compose(eq,R,R).            invert(X,X).
compose(R,eq,R).
compose(ne,ne,eq).          equal(eq).
```

Here is a very simple example of a query for Booleans. Note that we introduce `0` and `1` by `make` and add `union(0,ne,1)` to enforce that they are distinct. This suffices to solve this type of Boolean inequations and to simplify them thanks to union-find such that the relation (`eq` or `eq` or none) between variables can be found in quasi-constant time.

```
?- make(0),make(1),union(0,ne,1),
   make(A),make(B),union(A,eq,B),union(A,ne,0),union(B,eq,1).
root(A,2), B-eq->A, 0-ne->A, 1-eq->A.
```

The result of the query shows that `A` is also equal to `1`.

Related Work. It is well known that 2-SAT (conjunctions of disjunctions of at most two literals) [1] and Horn-SAT (conjunctions of disjunctions with at most one positive literal, i.e. propositional Horn clauses) [3,5,11] can be checked for satisfiability in linear time.

The class of Boolean equations and inequations we can deal with is a proper subset of 2-SAT, but not of Horn-SAT, since `A ne B` $\Leftrightarrow (\mathtt{A} \vee \mathtt{B}) \wedge (\neg\mathtt{A} \vee \neg\mathtt{B})$.

These two classical linear-time SAT algorithms assume that the graph is initially known, because it has to be traversed along its edges. The algorithms only check for satisfiability and can report one possible solution, but they do not simplify or solve the given problem in a general way.

The 2-SAT algorithm translates a given problem into a directed graph where arcs are the implications that are logically equivalent to the individual clauses in the problem. It then relies on a linear-time preprocessing of the given graph to find is maximal strongly connected components in reverse topological order. Then truth values are assigned to components by assigning them to all nodes in that component. Respecting the topological order, truth values are propagated through the components.

In contrast, our generalised union-find algorithm is an online algorithm and incremental in the sense of constraint logic programming. It can find the relation between two given variables in amortised quasi-constant time and space. It produces a simple normal form that has the same size of the original problem. By using find, the results can be normalised in quasi-linear time. It can be used for ask and tell, for assertion and entailment testing of constraints and is thus well-suited to be used in constraint solvers. It can even be run in parallel on variable-disjoint parts of the problem (that is, unions that share variables must be processed in sequential order). A more parallel version of union-find in CHR is discussed in [7].

Our Boolean algorithm instance can be integrated into a Boolean constraint solver. For example, the classical Boolean solver in CHR is based on value (unit) propagation, e.g. `and(X,Y,Z) <=> X=0 | Z=0`, and propagation of equalities, e.g. `and(X,Y,Z) <=> X=Y | Y=Z`. It can be now extended by propagation of inequalities, e.g. `and(X,Y,Z) <=>` $\mathtt{X\bar{Y}}$ `| Z=0` and `and(X,Y,Z) <=>` $\mathtt{X\bar{Z}}$ `| X=1,Y=0,Z=0`.

Can we extend our algorithm instance of generalised union-find to deal with 2-SAT? As put to use in the classical algorithm, any disjunction in two variables, $\mathtt{A} \vee \mathtt{B}$ can be written as implication $\neg\mathtt{A} \rightarrow \mathtt{B}$. Since we can get negation using an auxiliary variable `A ne negA`, we just would have to introduce the relation $\rightarrow$ (that corresponds to a total non-strict order $\leq$ on the truth values). But the implication relation looses too much information when composed. For example, given a tree `B-`$\leq$`->A, C-`$\leq$`->A`, `B` and `C` can be arbitrarily related. So if one adds `union(B,eq,C)` it has no effect on the tree, and thus the information that `B eq C` is lost.

5 Instance of Linear Polynomials

Another instance of our generalised union-find algorithm deals with linear polynomial equations in two variables. It features an infinite domain and an infinite number of relations. The CHR data constraint `X-A|B->Y` (with A≠0) now means `X=A*Y+B`. Note that these type of equations can be interpreted as functions. The operations on relations are defined as follows:

```
compose(A|B,C|D,A*C|A*D+B).
invert(A|B,1/A|-B/A).                    equal(1|0).
```

Again, a small example illustrates the behaviour of this instance.

```
?- make(X),make(Y),make(Z),make(W),
   union(X,2|3,Y),union(Y,0.5|2,Z),union(X,1|6,W).
root(X,1), Y-0.5|-1.5->X, Z-1.0|-7.0->X, W-1.0|-6.0->X.
```

Note that by the generic `linkEq` rule, `link(X,1|0,X)` will succeed but all other equations involving only one variable will fail. While this is as expected for e.g. `link(X,1|1,X)`, the equation `link(X,2|1,X)` has a solution `X=-1`. Indeed, in our program, failure will occur whenever a variable is fixed, i.e. determined to take a unique value. Our algorithm succeeds exactly when the set of equations has infinitely many solutions.

We now slightly modify the program code of the instance of our generalised union-find algorithm in order to introduce concrete numeric values and to solve for determined variables. Since there infinitely many numbers, we express them in terms of a single number, `1`. To make sure that the number `1` always stays the root, so that it can be always found by the find operation, we add `root(1,∞)` instead of `make(1)` to the beginning of a query. We replace the `linkEq` rule by the two rules. The first restricts applicability of the generic `linkEq` rule to the case where `A=1`, the second rule applies to equations that determine their variable and normalises the equation such that the coefficient is `1` and the second occurrence of the variable is replaced by the value `1`.

```
linkEq1 @ link(X,A|B,X) <=> A=:=1 | B=:=0.
linkEq2 @ link(X,A|B,X) <=> A=\=1 | link(X,1|B/(1-A)-1,1).
```

Note that there is a subtle point about these two rules: `X` may be the value `1`, and in that case the execution of `link(X,1|B/(1-A)-1,1)` in the right hand side of rule `linkEq2` will use rule `linkEq1` to check if `B/(1-A)-1` is zero (which holds if `B=1-A`).

The following tiny examples illustrate the behaviour of these two rules (∞ is chosen to be `9`):

```
?- root(1,9), make(X),make(Y),
   union(X,2|3,Y),union(X,4|1,1).
root(1,9), X-4|1->1, Y-0.5|-1.5->X.

?- root(1,9), make(X),make(Y),union(X,4|1,1),union(X,2|3,Y).
root(1,9), X-4|1->1, Y-2|-1->1.
```

We may add another rule that propagates values for determined variables down the tree data structure and so binds determined variables:

```
X-A|B->N <=> number(N) | X=A*N+B.

?- root(1,9), make(X),mak e(Y),
   union(X,2/3,Y),union(X,4/1,1).
root(1,9), X=5, Y=1.
```

Related Work. [2] gives a linear time algorithm that shares many principles with ours, but is more complicated. Equations correspond to directed arcs in a graph. Like the 2-SAT algorithm [1], it computes maximal strongly connected components. Inside each component, a modification of any linear-time spanning tree algorithm can be used to simplify the equations. The overall effect is the same as with our algorithm, and the algorithm is similar on the components, especially if Kruskal's algorithm [10] for spanning trees is used which relies on union-find. However, our algorithm is simpler and more general in its applicability. It does not need to compute strongly connected components or spanning tress, it directly uses union-find and moreover is incremental and parallelisable.

6 Complexity and Correctness

We want to show that our algorithm is a canonical extension of the optimised union-find algorithm in CHR. In other words, if we instantiate our algorithm to the case where the only relation is `=`, we get back the original program.

We first discuss *complexity* of our algorithm.

Theorem 1 (Complexity). Our algorithm has the same time and space complexity as the original algorithm if the operations on relations take constant time and space.

Proof Sketch. Any computation in our generalised algorithm can be mapped into a computation of the original union-find algorithm or it fails.

In the generalisation, we added arguments for the relations to existing CHR constraints that are variables and constraints on these variables only. The additional variables on the left hand side of each rule are all distinct and the guards have not been changed. The additional constraints on the right only constrain the new variables. In CHR, additional constraints can cause failure (inconsistency) or make more rules applicable, but never less. Since the new constraints are on new variables only, and these are not checked by the left hand sides and guards of rules, the applicability of the rules remains unchanged. So the only change is that a rule application could cause failure where the computation proceeded before (this applies to the `linkEq` rule).

More formally, we construct a mapping function that removes the additional arguments and additional built-in constraints. The claim is then shown by induction on length of the derivation and case analysis of the rules applicable in a derivation step. □

Next we show that our implementation is *correct*, if the involved relations are *bijective functions*. In that case, the composition operation is precise enough in that it allows to derive any of the three involved relations from the other two.

Intuitively the reason why we cannot work with arbitrary relations is that given n variables, there are up to $n(n-1)$ binary relations between different variables, but we can only represent $n-1$ of them in the tree data structure of the union-find algorithm. That means that all the other possibly existing relations must be computable from these $n-1$ relations.

In the following we assume w.l.o.g. that the domains of a variable include only those values for which the relations in which it occurs are defined. If the relation is a function this means that the function is total.

Definition 1. A function is *bijective* if for every y there is exactly one x such that $f(x) = y$, that is $f(x) = y \wedge f(u) = v \wedge (x = u \vee y = v) \rightarrow x = u \wedge y = v$.

Bijective functions are closed under inverse and composition.

While bijective functions may seem quite a strong restriction we remind the reader that permutations, isomorphisms and many other mappings (such as encodings in cryptography) are bijective functions and that most arithmetic functions are at least piecewise bijective, since they are piecewise monotone. Indeed, for a domain of size n, there exist $n!$ different bijective functions, i.e. more than exponentially many.

The identity function *id* is bijective. The relations `eq` and `ne` are bijective functions for domains with at most two values, and thus for Booleans. Linear polynomial equations in two variables can also be interpreted also bijective functions.

Theorem 2 (Correctness). The logical reading of the rules of our generalised union-find algorithm (for classical union-find, see [13]) is a consequence of a theory for the relations if these relations are bijective functions.

Proof Sketch. We prove by analysing the logical reading of the rules, where we replace `union`, `find`, `link` and `->` as intended by the binary relations between their variables (using infix notation), and the constraints for operations on relations by their definitions using functional notation. As usual, formulas are assumed to be universally closed.

```
(make)      make(X) ⇔ root(X,0).
(union)     (X XY Y) ⇔ ∃XA,A,YB,B,AB ((X XA A) ∧ (Y YB B) ∧
                          XA^-1∘XY∘YB=AB ∧ (A AB B))
```

```
(findNode)   (X XY Y) ∧ (X XR R) ⇔ ∃YR ((Y YR R) ∧
                         XY∘YR=XR ∧ (X XR R))
(findRoot)   root(X,N) ∧ (X XR R) ⇔ root(X,N) ∧ XR=id ∧ X=R

(linkEq)     (X XX X) ⇔ XX=id
(linkLeft)   RX>=RY ⇒ ((X XY Y) ∧ root(X,RX) ∧ root(Y,RY) ⇔
               ∃YX (XY^-1=YX ∧ (Y YX X) ∧ root(X,max(RX,RY+1))))
(linkRight)  RY>=RX ⇒ ((X XY Y) ∧ root(Y,RY) ∧ root(X,RX) ⇔
               (X XY Y) ∧ root(Y,max(RY,RX+1)))
```

Even though the logical reading in first order logical does not reflect the intended meaning of the `root` data constraint [13] (and a linear logic semantics is more faithful [4]), the logical reading suffices for our purposes.

Most rules lead to formulas that do not impose any restriction on the binary relations involved. The logical readings of `linkEq` and `findRoot` imply that the only relation that is allowed to hold between identical variables is the identity function *id*. The `findNode` rule, however, tells us a logical equivalence,

`(X XR R) ∧ (X XY Y) ⇔ (X XR R) ∧ (Y YR R)` where `XY∘YR=XR`,

that is not a tautology and restricts the involved relations. (For example, it does not hold for $\leq$`=XR=YR=XY` even though $\leq \circ \leq = \leq$.)

This conditions is obviously satisfied if the involved relations are bijective functions, because then, for any value given to one of the variables, the values for the other two variables are uniquely determined on both sides of the logical equivalence and there cannot be another triple of values that has any of the values in the same component. □

To further illustrate the Correctness Theorem, let `XR = id`. Then the above condition simplifies to: `(X XY Y) ⇔ (Y YR X)` where `XY∘YR=id`. This means that for any relation g that takes the place of `XY` or `YR`, there must be a right and a left inverse, i.e. $g \circ g^{-1} = g^{-1} \circ g = id$. This is the case if the relations are bijective functions. As a counter-example, take the function g that is defined by $g(a){=}c, g(b){=}c$ and its inverse g^{-1}. Now g is not bijective since for c there exists more than one value that yields it when g is applied to it. Also, g^{-1} is not a function. The composition $g^{-1} \circ g$ yields the universal relation that includes any pair taken from $\{a, b\}$. The composition $g \circ g^{-1}$ yields the pair $< c, c >$. But both should yield the identity function *id*.

Intuitively, we can replace any operation in our generalised union-find algorithm on a given bijective function f by a conjunction of operations on its individual values using the classical union-find algorithm. That is, we claim $X \in D_X \wedge Y \in D_Y \wedge \texttt{union}(X, f, Y)$ is equivalent to $\bigwedge_{X \in D_X} \texttt{union}(X, f(Y))$, where D_V denotes the domain of values for variable V.

7 Conclusions

In this extended abstract we have presented work in progress about extending the applicability of union-find implemented in CHR. We saw that the generalisation of the algorithm from maintaining equalities to certain binary relations (in particular bijective functions that admit precise composition) is straightforward in CHR and that the generalisation does not compromise quasi-linear time efficiency. We have implemented the generalisation and two instances, for equations and inequations over Booleans and for linear polynomial equations in two variables.

Our implementation in Sicstus Prolog CHR is available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/ufe.pl`.

While linear-time algorithms are known to check satisfiability and to exhibit certain solutions of these problems, our algorithms are simple instances of the generic algorithm and have additional properties that make them suitable for incorporation into constraint solvers: From classical union-find, they inherit simplicity, the possibility to both assert relations and test for entailed relations as well as quasi-linear time and space. By nature of CHR, they are anytime and online algorithms. They can be parallelised and solve and simplify the constraints in the problem, even when the constraints arrive incrementally, one after the other.

It remains to show that the proofs for complexity and correctness are as straightforward as claimed in this extended abstract. Future work will also try to extend the class of bijective functuions to other binary relations, and to investigate relationship with classes of tractable constraints. We also would like to investigate the potential tradeoff between efficiency and precision (i.e. by applying our generalised union-find to inequalities like $\leq$).

References

1. B. Aspvall, M.F. Plass, and R.E. Tarjan. A linear time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
2. Bengt Aspvall and Yossi Shiloach. A fast algorithm for solving systems of linear equations with two variables per equation. *Linear Algebra and its Applications*, 34:117–124, 1980.
3. Catriel Beeri and Philip A. Bernstein. Computational problems related to the design of normal form relational schemas. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.
4. Hariolf Betz and Thom Frühwirth. A linear-logic semantics for constraint handling rules. In P. van Beek, editor, *11th Conference on Principles and Practice of Constraint Programming CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 137–151. Springer, October 2005.
5. William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.
6. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1–3):95–138, 1998.
7. T. Frühwirth. Parallelizing union-find in constraint handling rules using confluence. In M. Gabbrielli and G. Gupta, editors, *Logic Programming: 21st International Conference, ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 113–127. Springer, October 2005.

8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming.* Springer, 2003.
9. Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
10. Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
11. Michel Minoux. LTUR: a simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. *Information Processing Letters*, 29(1):1–12, September 1988.
12. T. Schrijvers and T. Frühwirth. Optimal union-find in constraint handling rules, programming pearl. *Theory and Practice of Logic Programming (TPLP)*, 6(1), 2006.
13. Tom Schrijvers and Thom Frühwirth. Analysing the CHR Implementation of Union-Find. In *19th Workshop on (Constraint) Logic Programming (W(C)LP 2005).* Ulmer Informatik-Berichte 2005-01, University of Ulm, Germany, February 2005.
14. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The Computational Power and Complexity of Constraint Handling Rules. In *Second Workshop on Constraint Handling Rules, at ICLP05*, Sitges, Spain, October 2005.
15. Robert E. Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, 1984.