Quasi-Linear-Time Algorithms by Generalisation of Union-Find in CHR

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 incremental algorithms for solving certain propositional logic (SAT) problems and polynomial equations in two variables. In general, we prove that when the relations are bijective functions, our generalisation yields a correct algorithm. We also show that bijectivity is a necessary condition for correctness if the relations include the identity function.

The rules of our generic algorithm have additional properties that make them suitable for incorporation into constraint solvers: from classical union-find, they inherit a compact solved form and quasi-linear time and space complexity. By nature of CHR, they are anytime and online algorithms. They solve and simplify the constraints in the problem, and can test them for entailment, even when the constraints arrive incrementally.

1 Introduction

Constraint Handling Rules (CHR) [Frü98, FA03, Frü08] is a logical constraint-based concurrent committed-choice programming language consisting of guarded rules that rewrite conjunctions of atomic formulas. The classical optimal union-find algorithm [TvL84] can be implemented in CHR with best-known quasi-linear time complexity [SF06, SF05]. This result is not accidental, since the paper [SSD05] shows that every (RAM machine) algorithm with at least linear time complexity (an algorithm that at least reads all of its input), can be implemented in CHR with best known time and space complexity. Such a result is not known to hold 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 paper 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 correctness and efficiency. Our generalised union-find algorithm then maintains relations between elements under the operation of adding relations.

From classical optimal union-find, our generalised algorithm inherits amortised quasi-linear time and space complexity as well as the possibility to both assert relations (tell) and test for entailed (implied) relations (ask) as will be explained below.

CHR is used here as an effective general-purpose programming language for implementing classical algorithms. Still, by nature of CHR, our implementation is an *anytime (approximation) algorithm and online (incremental) algorithm.* Anytime algorithms can be stopped during their execution and exhibit an intermediate result from which to continue computation. In CHR, we can stop after any rule application and observe the intermediate result in the current state (store). Online algorithm means that the input (in CHR, the constraints of a query) can arrive incrementally, one after the other, without the need to recompute from scratch.

In the context of this paper, the algorithms are used in a classical way, i.e. with certain input to produce a desired output. We do not make use of the concurrent constraint programming features of CHR, where execution is also possible with incomplete and unknown inputs and where constraints - in this context, operations - are delayed until further information becomes available.

We can interpret the relations that we maintain as non-trivial equations. Under this point of view, our generalisation produces a *compact solved normal form* that represents all solutions of the given problem (a set of equations). The solution has at most the size of the original problem. We can also test the equations for entailment. All this is possible even when the constraints arrive incrementally. Hence our algorithm and its operation constraints are well-suited to be used inside constraint solvers.

Overview of the Paper. In the next two sections we quickly introduce CHR and then the union-find algorithm and its implementation in CHR. In Section 4 we generalise the union-find algorithm. The next section proves optimal time and space complexity of our generalisation. We then show correctness in Section 6 when the involved relations are bijective functions. We present two instances in Sections 7 and 8. We end with conclusions. This paper is a revised and extended version of the extended abstract [Frü06].

2 Constraint Handling Rules (CHR)

CHR [Frü98, FA03, Frü08] manipulates conjunctions of constraints (relations, predicates, atoms) that reside in a constraint store. In the following, the meta-variables H, G, B and C denote conjunctions of constraints, denoting head (parts of the head), guard, body of a rule and constraints from a state, respectively. In a CHR program, the set of constraints defined in the heads and the set of the constraints used in the guards are disjoint. The former are called CHR constraints. The latter are called built-in constraints and their meaning is defined by a logical theory named CT. Standard built-in constraints are `true`, `false`, as well as syntactic equality `=` and arithmetic equality `=:=`.

CHR programs are composed of two main types of rules as given in Fig. 1. A third, hybrid kind called simpagation rules is not essential for this paper. In the figure, we also give the declarative, logical reading (meaning) of the rules, where $\bar{y}$ are the variables that appear only in the body B of a rule. W.l.o.g. we assume for simplicity that variables that appear both in the guard and body also appear in the head of a rule. A simplification rule corresponds to a logical equivalence provided the guard holds, while a propagation rule corresponds to an implication.

Simplification rule:	$H \Leftrightarrow G \mid B$	$\forall \bar{x}\ (G \rightarrow (H \leftrightarrow \exists \bar{y}\ B))$
Propagation rule:	$H \Rightarrow G \mid B$	$\forall \bar{x}\ (G \rightarrow (H \rightarrow \exists \bar{y}\ B))$

Fig. 1. Main Types of CHR Rules and their Logical Reading

The *standard (abstract) operational semantics* of CHR is given by a transition system where states are conjunctions of constraints. In Figure 2 we just give the transition for simplification rules, since we only need this kind of rules in this paper. For the propagation rules the transition is very similar, the only difference is that the head is kept. In the transition system, CHR constraints are treated

if $H \Leftrightarrow G \mid B$ is a copy of a rule $H \Leftrightarrow G \mid B$ with new variables $\bar{X}$
and $CT \models \forall(C \rightarrow \exists \bar{X}(H{=}H' \wedge G))$
then $(H' \wedge C) \longmapsto (B \wedge H{=}H' \wedge C)$

Fig. 2. State transition for simplification rules

on a syntactic level, while built-in constraints are treated on a semantic level using logic. A simplification rule replaces instances of the CHR constraints H by B provided the guard test C holds. A propagation rule instead just adds B to H without removing anything.

The constraints of the store comprise the *state* of an execution. Starting from an arbitrary initial store (called *query*), CHR rules are applied exhaustively until a fixpoint is reached. The resulting sequence of state transitions is called a *computation.* A rule is applicable, if its head constraints are matched by constraints in the current store one-by-one and if, under this matching, the guard of the rule is logically implied by the constraints in the store. This applicability condition is formally defined by the formula $CT \models \forall(C \rightarrow \exists \bar{X}(H{=}H' \wedge G))$ in the transition system, where CT is the logical theory for the constraints used in guards of the rules. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice.

The standard semantics is too abstract to describe the details of CHR implementations. For this purpose, an instance of the standard semantics, called *refined semantics* was formalised in [DSdlBH04]. Our complexity proof relies on this refined semantics. We will shortly describe this semantics. It is, however, beyond the scope of the paper to present the details of this semantics.

Queries are executed from left to right and for each new constraint, rules are applied top-down in the textual reading order of the program. Trivial non-termination of propagation rule applications is avoided by applying them at most once to the same constraints. Built-in constraints in the store are simplified and solved, and that in particular variables that are constrained to take a unique value are equated with that value.

In this refined semantics of actual implementations, a CHR constraint in a query can be understood as a procedure that goes efficiently through the rules of the program in the order they are written, and when it matches a head constraint of a rule, it will look for the other, *partner constraints* of the head in the constraint store and check the guard until an applicable rule is found. We consider such a constraint to be *active*. If the active constraint has not been removed after trying all rules, it will be put into the constraint store. Constraints from the store will be reconsidered (woken) if newly added built-in constraints constrain variables of the constraint, because then rules may become applicable since their guards are now implied. In particular this will be the case if a syntactic or arithmetic equality binds a variable to a constant or another variable.

3 The Union-Find Algorithm

In this section we follow the exposition of [SF06]. The classical union-find (also referred to as disjoint-set-union) algorithm was introduced by Tarjan in the seventies [TvL84]. A classic survey on the topic is [GI91]. 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)`.

3.1 Implementing Union-Find in CHR

In the naive union-find algorithm without optimisations, the 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, we `link` them by making one root point to the other root.

The following CHR program implements the operations and data structures of the naive union-find algorithm. The CHR constraints `make/1`, `union/2`, `find/2` and auxiliary `link/2` define the corresponding 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 infix notation for `->/2` to evoke the image of a pointer (directed arc).

```
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).
```

In the rules `findNode` and `findRoot`, the data constraints `X->Y` and `root(X)`, respectively, occur in the head and body of their rules. In a naive CHR implementation, when the rule applies, they will be removed and immediately re-added again. For the programs discussed in this paper, this causes only constant time overhead. In general, the simpagation rule notation as well as optimising CHR compilers avoid this overhead by leaving the constraint in the store.

3.2 Optimised Union-Find

The basic algorithm requires $\mathcal{O}(n)$ time per find (and union) operation in the worst case, where n is the number of elements (and thus of 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 and balanced by pointing the root of the smaller tree to the root of the larger tree without changing its rank. *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 and the rank is incremented by one. With this optimisation, the height of the tree can be logarithmically bound.

The following CHR program implements the resulting optimised classical union-find algorithm with path compression for find and union-by-rank [TvL84].

```
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. In the body of the rule, the order of constraints find(Y,R), X->R optimises execution under the refined semantics of CHR, since under left-to-right execution, the pointer constraint is only introduced when R has been computed by find. The link rule has been split into two rules, linkLeft and linkRight, to reflect the optimisation of union-by-rank.

4 Generalised Union-Find

The idea of generalising union find is to replace equations between variables by binary relations. Our generalised union-find algorithm then maintains relations between elements under the operation of adding relations. The operation constraints union, find, link and the data constraint -> get an additional argument to hold the relation. 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 stores the relation in the tree data constraint. The arcs in the tree are labeled by relations now.

We assume that in queries, all relations are given. While the program in principle would also compute with unknown relations, we are not interested in these computations in the context of this paper. Remember that a query contains only make, union and find operations. A new element must be introduced first by a single make before subjected to union and find.

We need some standard *operations on relations* from relational algebra and a non-standard one, combine. The operations are implemented by constraints as follows, where *id* is the identity function:

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

- `equal`(r_1) iff $r_1 = id$
- `combine`(r_1, r_2, r_3, r_4) iff $r_4 := r_1^{-1} \circ r_3 \circ r_2$

The commutative diagram below shows the relations between the four relations that are arguments of `combine`. The question mark after `AB` reminds us that this relation is the one that `combine` computes from the other three relations.

```
X -- R1 -- A
|          |
R3         R4?
|          |
Y -- R2 -- B
```

The following code extends the CHR implementation of optimal union-find by additional arguments (the relations) and by additional constraints on them (the operations on relations). These additions are in italics for clarity. Our implementation in the Sicstus 3 Prolog CHR library is available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/ufe.pl` and can be run with CHR online at `chr.informatik.uni-ulm.de/~webchr/webchr1`.

```
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. In the union, `combine` computes the relation `AB` that must hold between the roots that are to be linked from the initial relation `XY` and the relations `XA` and `YB` resulting from the two find operations.

Note that in the `linkEq` rule, the link operation now tests if the relation `XX` given between two identical variables `X` is the identity relation. If this is not the case, the overall computation will stop with an inconsistency error. This happens for example, if we try to union the same two variables twice with different incompatible relations between them.

The choice of the operations on relations added in the program is justified by the intended correctness with regard to the logical reading of the program as discussed in Section 6. Since all relations in a query are given, and since then the `find` operation returns a relation, the operations on relations `compose`, `combine` and `invert` compute a relation (the last argument) from given relations and `equal` checks a given relation.

In the body of the rules the sequence of the constraints takes advantage of the left-to-right execution order of the refined CHR semantics. The order of constraints is the same as of the corresponding operations in a procedural programming language.

Normalisation and Entailment Checking

By using the find operation, the results can be further normalised: for each variable in the problem, we issue a find operation. It will return the relation to the root variable and as a side-effect the tree will be compressed to have a direct pointer between the two variables. So afterwards, the solved form contains data constraints of the form X_i`-XR->`R_j, where all X_i are different (and the R_j are root variables only).

Also we can *check for entailment*, i.e. ask if a given relation holds between two given variables. This is the case if their roots are the same (otherwise they are unrelated) and if asserting the relation using union would not change the tree. That is, a given relation already holds between two variables if the union operation leads to a link operation that does not update the tree. This is the case if the `linkEq` rule is applicable. Therefore the following special instance of the `union` rule suffices for entailment of a relation `(X XY Y)`:

```
unioned?(X,XY,Y) <=>
 find(X,XA,A),find(Y,YB,B), combine(XA,YB,XY,AB), A==B, equal(AB).
```

`A==B` checks if `A` and `B` are identical and `equal(AB)` checks if `AB` is the identity relation. These checks are inherited from the `linkEq` rule.

The query associated with the entailment test can also be modified to find out what relation between two given variables `X` and `Y` holds. In that case we replace `combine(XA,YB,XY,AB)` according to its definition so that it computes `XY` from `XA`, `YB` and `AB` (which must be identity). We get the rule:

```
related?(X,XY,Y) <=>
 find(X,XA,A),find(Y,YB,B), A==B, invert(YB,BY),compose(XA,BY,XY).
```

where just `X` and `Y` are given.

5 Complexity

Our algorithm is a canonical extension (proper generalisation) of the optimised union-find algorithm in CHR: we added arguments holding the relations to existing CHR constraints. In the rules, these additional arguments for the relations are variables. In the head of each rule, these variables are all distinct. The guards have not been changed. The additional constraints in the rule bodies only involve variables for relations. These operations on relations can be performed in constant time. Moreover, if we specialise our algorithm to the case where the only relation is identity `id`, we get back the original program. These observations are an indication that we can preserve the complexity and correctness results of the original algorithm under certain conditions.

We first have to establish some general lemmas for CHR rule applications. We restrict ourselves to simplification rules in this section, since this is the only type of rules we use in this paper. Then, our proof is based on a mapping from computations in our generalised algorithm to computation in the original union-find algorithm.

Definition 1. Given a CHR simplification rule $R = H \Leftrightarrow G \mid B$, then $(H \wedge G)$ is called the *minimal state of* R and the transition $(H \wedge G) \longmapsto (B \wedge G)$ is called the *minimal transition of* R.

Minimal transitions capture the essence of a CHR rule application.

Lemma 1 (Minimal Transitions). A CHR simplification rule $R = H \Leftrightarrow G \mid B$ is applicable to its minimal state, $(H \wedge G)$, leading to the minimal transition of R, $(H \wedge G) \longmapsto (B \wedge G)$.

Proof. The proof is straightforward from the operational semantics of CHR given as transition system in Figure 2. We take a copy of the rule R, $H' \Leftrightarrow G' \mid B'$. It obviously satisfies the rule applicability condition of the transition, $CT \models \forall(G \rightarrow \exists\bar{X}(H'{=}H \wedge G'))$, where $H'{=}H$ is simply a variable renaming that makes G and G' equivalent. We can apply this variable renaming in the resulting state $(B' \wedge H'{=}H \wedge G)$ to get $(B \wedge G)$. $\square$

Lemma 2 (Arbitrary Transitions). Any transition $(H' \wedge C) \longmapsto (B \wedge H{=}H' \wedge C)$ resulting from application of a rule R (cf. Fig. 2) can be derived from the minimal transition $(H \wedge G) \longmapsto (B \wedge G)$ of R (cf. Lemma 1) by instantiating variables according to the equation $H{=}H'$ and by replacing the guard constraints G with $H{=}H' \wedge C$.

Proof. Based on the transition system in Fig. 2, we only have to note that the source states $(H \wedge H{=}H' \wedge C)$ and $(H' \wedge C)$ are identical: By definition H and $(H' \wedge C)$ do not have any variables in common, therefore we can replace H by H' and remove the redundant resulting equation $H'{=}H'$. $\square$

In our algorithm, rule applications take constant time as in the original union-find algorithm in CHR if the additional operations on relations take constant time.

Lemma 3 (Union-Find Rule Application Complexity). Every rule of the generalised union-find algorithm can be applied in constant time and space under the refined semantics of CHR [DSdlBH04], if the introduced operations on relations (`combine, compose, equal, invert`) take constant time and space.

Proof Sketch. The proof is essentially the same as that for the original optimal union-find algorithm as presented in Section 6 of [SF06]. We only repeat the main points here.

Following the refined semantics of CHR [DSdlBH04], CHR implementations exist where all of the following take constant time [SF06]:

– finding all constraints with a particular value in a given argument position (due to indexing),
– matching of constants and variables in the head of a rule,
– testing and solving simple built-in constraints (like `=`, `=<` and `>=`),
– adding and deleting CHR constraints.

It is further assumed that storing the (data) constraints and their indexes takes constant space per constraint and variable.

From the above assumptions it is shown that processing a data constraint under the refined semantics takes constant time: the constraint is called, some rules are tried, some partner constraints which share a variable with the active constraint are looked for, but none are present, and finally the call ends with inserting the data constraint into the constraint store. It follows that all rule tries and applications with an active constraint take constant time. □

We now can show optimal time and space complexity of our algorithm.

Theorem 1 (Optimal Complexity). Our generalised union-find algorithm in CHR has the same time and space complexity as the original optimised union-find algorithm if the introduced operations on relations (`combine, compose, equal, invert`) take constant time and space.

Proof Sketch. We prove the Theorem by showing that any computation in our generalised algorithm can be mapped into a computation of the original union-find algorithm with the same time and space complexity. The claim is shown by induction on length of the computation and case analysis of the rules applicable in a computation step. It is thus sufficient to consider individual rule applications. Since each rule application takes constant time and space in both algorithms by Lemma 3, it suffices to show that the computations lengths are linearily related.

We construct a function that maps transitions of our generalised union-find algorithm to transitions of the optimised union-find algorithm. The mapping function τ simply removes the additional arguments holding the relations and additional constraints for the operations on the relations. The function τ is exhaustively defined by the following equalities:

$\tau(\texttt{(A} \longmapsto \texttt{B))} = \tau(\texttt{A}) \longmapsto \tau(\texttt{B})$
$\tau(\texttt{(A} \wedge \texttt{B))} = \tau(\texttt{A}) \wedge \tau(\texttt{B})$
$\tau(\texttt{make(X)}) = \texttt{make(X)}$
$\tau(\texttt{union(X,XY,Y)}) = \texttt{union(X,Y)}$
$\tau(\texttt{find(X,XY,Y)}) = \texttt{find(X,Y)}$
$\tau(\texttt{link(X,XY,Y)}) = \texttt{link(X,Y)}$
$\tau(\texttt{X=Y}) = \texttt{true}$ if `X` and `Y` are relations
$\tau(\texttt{X=Y}) = \texttt{X=Y}$ if `X` and `Y` are elements
$\tau(\texttt{X>=Y}) = \texttt{X>=Y}$
$\tau(\texttt{root(X,Y)}) = \texttt{root(X,Y)}$

$\tau($`X-XY->Y)`$) =$ `X->Y`
$\tau($`combine(XA,YB,XY,AB)`$) =$ `true`
$\tau($`compose(XY,YR,XR)`$) =$ `true`
$\tau($`equal(XX)`$) =$ `true`
$\tau($`invert(XY,YX)`$) =$ `true`

Note that the constraint `true` is the neutral element for conjunction, i.e. `true` ∧ `A` and `A` ∧ `true` are each the same as `A`.

We now establish correctness of the mapping: in our generic algorithm, transitions can be caused by the application of the union-find rules or of rules that define the operations on relations (`combine, compose, equal, invert`) for the specific instance. For the union-find rules, we have to show that for each rule application in a transition of our generic algorithm, there is an application of the rule by the same name in the corresponding mapped transition in the original union-find program.

We need not apply rules to arbitrary states, but just have to consider minimal transitions by Lemma 2. It remains to show that the function τ correctly maps the queries and the minimal transitions for each rule of union-find. For the `union` rule we have that

```
τ(union(X,XY,Y) ⟼ find(X,XA,A) ∧ find(Y,YB,B) ∧
                   combine(XA,YB,XY,AB) ∧ link(A,AB,B)) =
  union(X,Y) ⟼ find(X,A) ∧ find(Y,B) ∧ link(A,B).
```

Analogously, the mapping can be applied to the other rules of the program.

If one of the rules for operations on relations is applied in a transition, the source and target state in the mapped transition are identical, because τ maps these operations and the built-in syntactic equalities for relations resulting from them all to `true`. Since we required that these operations take constant time and space, each operation can only cause a constant number of transitions. Therefore their execution causes only a constant time overhead for each rule of our algorithm.

We also have to consider inconsistency errors caused by execution of `equal` caused by the application of a `linkeq` rule. In that case, the computation stops in our generic algorithm, while in the corresponding mapped transition, `equal` is mapped to true and the computation can possibly proceed. Clearly it needs less time and space to stop a computation early. □

6 Correctness

By correctness of a program we mean that the logical reading of the rules of a program is a logical consequence of a specification given as a logical theory. Since our generalised union-find algorithm maintains relations between elements under the operation of adding relations, the specification is a theory for these relations. Since our program should work with arbitrary relations, we expect the logical reading of its rules to follow from the empty theory, i.e. to be tautologies. We will see that this is not the case for all rules. To this end, we prove that when the relations involved are bijective functions, our generalisation yields a correct algorithm. We also show that bijectivity is a necessary condition for correctness if the relations include the identity function.

In the logical reading of our rules, 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. Even though the logical reading of union-find does not reflect the intended meaning of the `root` data constraint [SF05], the logical reading suffices for our purposes.

```
(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)))
```

Most rules lead to formulas that do not impose any restriction on the binary relations involved. However, the logical reading of `linkEq` and `findRoot` implies that the only relation that is allowed to hold between identical variables is the identity function *id*. Most importantly, the meaning of the `findNode` rule is a logical equivalence, that is not a tautology and restricts the involved relations. For example, it does not hold for $\leq$`=XR=YR=XY` even though $\leq \circ \leq = \leq$.

We now 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. For most other types of relations, our generalised union-find algorithm is not correct, since it looses information due to composition.

Definition 2. A function f is *bijective* if the function is injective and surjective, i.e. $f(\bar{x}) = y \wedge f(\bar{u}) = v \wedge (\bar{x} = \bar{u} \vee y = v) \rightarrow \bar{x} = \bar{u} \wedge y = v$.

Thus a unary function f is bijective if for every x there is exactly one y and vice versa such that $f(x) = y$. Bijective functions are closed under inverse and composition.

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

Proof. The identity function `id` needed by rules `findRoot` and `linkEq` is a bijective function. The `findNode` rule leads to the non-tautological formula,

`(X XR R)` $\wedge$ `(X XY Y)` $\Leftrightarrow$ `(X XR R)` $\wedge$ `(Y YR R)` where `XY`$\circ$`YR=XR`.

This condition 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 (x, y, z) that has any of the values in the same component. All other rules are tautologies. □

Next we show that when the identity function `id` is one of the relations, then all relations must be bijective.

Theorem 3 (Bijectiveness). The logical reading of the rules of our generalised union-find algorithm implies that all relations are bijective if the allowed relations include the identity function.

Proof. In the formula for rule `findNode`,

`(X XR R)` $\wedge$ `(X XY Y)` $\Leftrightarrow$ `(X XR R)` $\wedge$ `(Y YR R)` where `XY`$\circ$`YR=XR`,

we consider two cases in which we replace either relation `XY` or relation `YR` by the identity function `id`. This leads to the two formulas

`(X XR R)` $\wedge$ `(X id Y)` $\Leftrightarrow$ `(X XR R)` $\wedge$ `(Y XR R)` and
`(X XR R)` $\wedge$ `(X XR Y)` $\Leftrightarrow$ `(X XR R)` $\wedge$ `(Y id R)`.

The former formula means that any relation `XR` used must be surjective, the latter means that any relation `XR` must be injective. Hence any relation must be bijective. □

The two instances of our generalised union-find algorithm that we will discuss next involve bijective functions only.

7 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 thus a (small) 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. Note that the `ne` relation holds if the Boolean exclusive-or function (`xor`) returns true. The operations on relations can be defined by the following rules:

```
compose(eq,R,S) <=> S=R.           invert(R,S) <=> S=R.
compose(R,eq,S) <=> S=R.
compose(R,R,S) <=> S=eq.           equal(S) <=> S=eq.

combine(XA,YB,XY,AB) <=>
    compose(XY,YB,XB), invert(XA,AX), compose(AX,XB,AB).
```

Here is a simple example of a query for Booleans. Note that we introduce the truth values `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.

```
?- 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`. More examples are available online.

Related Work. It is well known that 2-SAT (conjunctions of disjunctions of at most two literals) [APT79] and Horn-SAT (conjunctions of disjunctions with at most one positive literal, i.e. propositional Horn clauses) [BB79, DG84, Min88] 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 are not incremental. They assume that the problem and its graph representation are 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, so the results are less informative than ours.

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 graph to find is maximal strongly connected components in reverse topological order. Respecting the topological order, truth values are propagated through the components, where all nodes in a component are assigned to the same truth value.

In contrast, our generalised union-find algorithm produces a simple normal form representing all solutions. Due to the properties of the generalised union-find algorithm, our Boolean instance can be integrated into a Boolean constraint solver. For example, the classical Boolean solver in CHR is based on value (unit) propagation, with rules such as `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) <=> X ne Y | Z eq 0` and `and(X,Y,Z) <=> X ne Z | X eq 1, Y eq 0, Z eq 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 implement negation using an auxiliary variable, e.g. `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. If one now asserts `union(B,eq,C)`, it has no effect on the tree, and thus the information that `B eq C` is lost.

8 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. In this instance, the CHR data constraint `X-A#B->Y` (with `A`$\neq$`0`) means `X=A*Y+B`. The operations on relations are defined as follows:

```
compose(A#B,C#D,S) <=> S = A*C # A*D+B.
invert(A#B,S) <=> S = 1/A # -B/A.           equal(S) <=> S = 1#0.

combine(XA,YB,XY,AB) <=>
             compose(XY,YB,XB), invert(XA,AX), compose(AX,XB,AB).
```

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 the generic `linkEq` rule asserts that the relation `XX` in `link(X,XX,X)` must be the identity function. Thus `link(X,1#0,X)` is fine, but all other equations of the form `link(X,A#B,X)` with `A#B` different from `1#0` will lead to an inconsistency error. While this is correct for `link(X,1#1,X)`, the equation `link(X,2#1,X)` should not fail as it does, since it has the solution `X=-1`. Indeed, in our program, an inconsistency will occur whenever a variable is fixed, i.e. determined to take a unique value. Our implementation succeeds exactly when the set of equations has infinitely many solutions.

We now introduce concrete numeric values and solve for determined variables. We express numbers as multiples of the 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 split the `linkEq` rule into two rules. The first restricts applicability of the generic `linkEq` rule to the case where `A=1`, the second rule applies otherwise, i.e. to equations that determine their variable (`A=\=1`) and normalises the equation such that the coefficient is `1` and the second occurrence of the variable is replaced by `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 small examples illustrate the behaviour of the two new 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.
```

The queries and answers mean the same, but the answers are syntactically different due to the different order of `union` operations in the query.

We add another rule that propagates values for determined variables down the tree data structure and so binds all determined variables in linear time:

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

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

More examples are available online.

Related Work. [AS80] gives a linear time algorithm that is similar to ours, but is more complicated. Equations correspond to directed arcs in a graph. Like the 2-SAT algorithm [APT79], it computes maximal strongly connected components, and thus the problem has to be known form the beginning. 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 [Kru56] 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.

9 Conclusions

We systematically extended the applicability of union-find algorithm as 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 and space efficiency. We have implemented the generalisation and two instances, for equations and inequations over Booleans and for linear 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 our generic algorithm. Our implementation in the Sicstus 3 Prolog CHR library is available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/more/ufe.pl` and can be run at `chr.informatik.uni-ulm.de/~webchr/webchr1`.

Our generic algorithm has desirable properties that make it suitable for incorporating its instances into constraint solvers: by nature of CHR, our implementation is an anytime algorithm and online algorithms. The rules solve and simplify the constraints in the problem, and can test them for entailment, even when the constraints arrive incrementally, one after the other.

From classical optimal union-find, our generic algorithm inherits amortised quasi-linear time and space complexity as well as the possibility to both assert relations and test for entailed relations. It produces a compact solved normal form that represents all solutions of the given problem and has at most the size of the original problem. By using the find operation, the results can be further normalised in quasi-linear time. The relation between two given variables can be found in quasi-constant time using find operations.

We have proven that when the relations involved are bijective functions, our generalisation yields a correct algorithm. We also showed that bijectivity is a necessary condition for correctness if the relations include the identity function. 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. Indeed, for a domain of size n, there exist $n!$ different bijective functions, i.e. more than exponentially many. Also, most arithmetic functions are at least piecewise bijective, since they are piecewise monotone.

Future work will try to extend the class of bijective functions to other binary relations by abandoning the identity function, and investigate the relationship with classes of tractable constraints. We also would like to find out about the potential tradeoff between efficiency and precision (i.e. when applying our generalised union-find to inequalities like $\leq$).

References

[APT79] Aspvall, B., Plass, M.F., Tarjan, R.E.: A linear time algorithm for testing the truth of certain quantified Boolean formulas. Information Processing Letters 8, 121–123 (1979)

[AS80] Aspvall, B., Shiloach, Y.: A fast algorithm for solving systems of linear equations with two variables per equation. Linear Algebra and its Applications 34, 117–124 (1980)

[BB79] Beeri, C., Bernstein, P.A.: Computational problems related to the design of normal form relational schemas. ACM Trans. Database Syst. 4(1), 30–59 (1979)

[DG84] Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional horn formulae. J. Log. Program. 1(3), 267–284 (1984)

[DSdlBH04] Gregory, J., Duck, P.J.: Stuckey, Maria Garcia de la Banda, and Christian Holzbaur. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132. Springer, Heidelberg (2004)

[FA03] Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer, Heidelberg (2003)

[Frü98] Frühwirth, T.: Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. Journal of Logic Programming 37(1–3), 95–138 (1998)

[Frü06] Frühwirth, T.: Deriving linear-time algorithms from union-find in chr. In: Schrijvers, T., Frühwirth, T. (eds.) Third Workshop on Constraint Handling Rules, Venice, Italy (July 2006)

[Frü08] Frühwirth, T.: Constraint Handling Rules. Cambridge University Press, Cambridge (to appear, 2008)

[GI91] Galil, Z., Italiano, G.F.: Data Structures and Algorithms for Disjoint Set Union Problems. ACM Comp. Surveys 23(3), 319ff (1991)

[Kru56] Joseph, B., Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society 7, 48–50 (1956)

[Min88] Minoux, M.: LTUR: a simplified linear-time unit resolution algorithm for Horn formulae and computer implementation. Information Processing Letters 29(1), 1–12 (1988)

[SF05] Schrijvers, T., Frühwirth, T.: 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)

[SF06] Schrijvers, T., Frühwirth, T.: Optimal union-find in constraint handling rules, programming pearl. Theory and Practice of Logic Programming (TPLP) 6(1) (2006)

[SSD05] Sneyers, J., Schrijvers, T., Demoen, B.: The Computational Power and Complexity of Constraint Handling Rules. In: Second Workshop on Constraint Handling Rules, at ICLP 2005, Sitges, Spain (October 2005)

[TvL84] Tarjan, R.E., van Leeuwen, J.: Worst-case Analysis of Set Union Algorithms. J. ACM 31(2), 245–281 (1984)