Using CHR to Derive More Linear-Time Algorithms from Union-Find

Thom Frühwirth

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

CHR 2006, Venice, July '06

Motivation

Constraint Handling Rules (CHR): logical concurrent committed-choice guarded rules with built-in constraints.

Classical optimal union-find algorithm [Tarjan+, JACM 31(2)] implementable in CHR with best-known quasi-linear time complexity [Schrijvers/Frühwirth, WCLP'05,ICLP'05,TPLP'06].

Can we use this efficient algorithm for more than disjoint set union and maintaing equality?

Can CHR help us in generalising union-find?

Do we get any new useful algorithms out of it?

Outline

- Constraint Handling Rules (CHR)
- Quasi-Linear Time Union-Find Algorithm
- Generalised Union-Find in CHR
- Instances of Boolean Inequations and Polynomial Equations
- Complexity and Correctness

Constraint Handling Rules (CHR)

- Constraint programming language for Computational Logic
- Multi-headed guarded committed-choice rules
  transform multi-set of constraints until exhaustion
- Ideal for executable specifications and rapid prototyping
- Implements algorithms with optimal time and space complexity
- Incrementality (on-line, any-time) and concurrency for free
- Logical and operational semantics coincide strongly
- High-level supports program analysis and transformation:
  Confluence/completion, termination/time complexity, correctness...
- Implemenations in most Prolog systems, Java, Haskell
- 100s of applications from types, time tabling to cancer diagnosis

Example Partial Order Constraint

$X \leq X$	$\Leftrightarrow$	$true$	(reflexivity)
$X \leq Y \wedge Y \leq X$	$\Leftrightarrow$	$X = Y$	(antisymmetry)
$X \leq Y \wedge Y \leq Z$	$\Rightarrow$	$X \leq Z$	(transitivity)

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A$$
$$\downarrow \quad \text{(transitivity)}$$
$$A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C}$$
$$\downarrow \quad \text{(antisymmetry)}$$
$$A \leq B \wedge B \leq C \wedge \underline{A = C}$$
$$\downarrow \quad \text{(built-in solver)}$$
$$\underline{A \leq B} \wedge \underline{B \leq A} \wedge A = C$$
$$\downarrow \quad \text{(antisymmetry)}$$
$$A = B \wedge A = C$$

Example Partial Order Constraint

$$X \leq X \Leftrightarrow true \quad \text{(reflexivity)}$$
$$X \leq Y \wedge Y \leq X \Leftrightarrow X = Y \quad \text{(antisymmetry)}$$
$$X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z \quad \text{(transitivity)}$$

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A$$
$\downarrow$ (transitivity)
$$A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C}$$
$\downarrow$ (antisymmetry)
$$A \leq B \wedge B \leq C \wedge \underline{A = C}$$
$\downarrow$ (built-in solver)
$$\underline{A \leq B} \wedge \underline{B \leq A} \wedge A = C$$
$\downarrow$ (antisymmetry)
$$A = B \wedge A = C$$

Example Partial Order Constraint

$$X \leq X \Leftrightarrow true \quad \text{(reflexivity)}$$
$$X \leq Y \wedge Y \leq X \Leftrightarrow X = Y \quad \text{(antisymmetry)}$$
$$X \leq Y \wedge Y \leq Z \Rightarrow X \leq Z \quad \text{(transitivity)}$$

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A$$
$$\downarrow \quad \text{(transitivity)}$$
$$A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C}$$
$$\downarrow \quad \text{(antisymmetry)}$$
$$A \leq B \wedge B \leq C \wedge \underline{A = C}$$
$$\downarrow \quad \text{(built-in solver)}$$
$$\underline{A \leq B} \wedge \underline{B \leq A} \wedge A = C$$
$$\downarrow \quad \text{(antisymmetry)}$$
$$A = B \wedge A = C$$

Example Partial Order Constraint

$X \leq X$	$\Leftrightarrow$	$true$	(reflexivity)
$X \leq Y \wedge Y \leq X$	$\Leftrightarrow$	$X = Y$	(antisymmetry)
$X \leq Y \wedge Y \leq Z$	$\Rightarrow$	$X \leq Z$	(transitivity)

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A$$

$\downarrow$ (transitivity)

$$A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C}$$

$\downarrow$ (antisymmetry)

$$A \leq B \wedge B \leq C \wedge \underline{A = C}$$

$\downarrow$ (built-in solver)

$$\underline{A \leq B} \wedge \underline{B \leq A} \wedge A = C$$

$\downarrow$ (antisymmetry)

$$A = B \wedge A = C$$

Example Partial Order Constraint

$X \leq X$	$\Leftrightarrow$	$true$	(reflexivity)
$X \leq Y \wedge Y \leq X$	$\Leftrightarrow$	$X = Y$	(antisymmetry)
$X \leq Y \wedge Y \leq Z$	$\Rightarrow$	$X \leq Z$	(transitivity)

$$\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A$$

$\downarrow$ (transitivity)

$$A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C}$$

$\downarrow$ (antisymmetry)

$$A \leq B \wedge B \leq C \wedge \underline{A = C}$$

$\downarrow$ (built-in solver)

$$\underline{A \leq B} \wedge \underline{B \leq A} \wedge A = C$$

$\downarrow$ (antisymmetry)

$$A = B \wedge A = C$$

Example Partial Order Constraint

$$
\begin{array}{rcll}
X \leq X & \Leftrightarrow & true & \text{(reflexivity)} \\
X \leq Y \wedge Y \leq X & \Leftrightarrow & X = Y & \text{(antisymmetry)} \\
X \leq Y \wedge Y \leq Z & \Rightarrow & X \leq Z & \text{(transitivity)}
\end{array}
$$

$$
\begin{array}{cl}
\underline{A \leq B} \wedge \underline{B \leq C} \wedge C \leq A & \\
\downarrow & \text{(transitivity)} \\
A \leq B \wedge B \leq C \wedge \underline{C \leq A} \wedge \underline{A \leq C} & \\
\downarrow & \text{(antisymmetry)} \\
A \leq B \wedge B \leq C \wedge \underline{A = C} & \\
\downarrow & \text{(built-in solver)} \\
\underline{A \leq B} \wedge \underline{B \leq A} \wedge A = C & \\
\downarrow & \text{(antisymmetry)} \\
A = B \wedge A = C &
\end{array}
$$

Union-Find Algorithm

Maintain disjoint sets under set union.
- Sets implemented as trees, nodes are set elements.
- Root is representative of the set.
- Union updates root, thus changes representative.

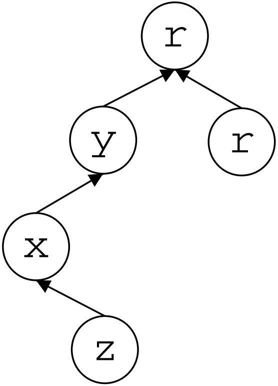
Operations:

- make(X): generate new tree with root node X.
- find(X,R): follow path from node X to root.
  Return root as representative R.
- union(X,Y): find representatives of X and Y.
  link them by making one point to the other.

Query: sequence of make and union operations.
Each node introduced by make. Nodes are variables or constants.

Basic Union-Find

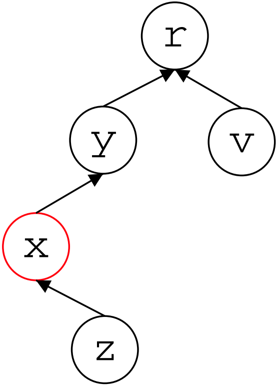
`find(x) =`



Tree graphics borrowed from Tom Schrijvers by kind permission.

Basic Union-Find

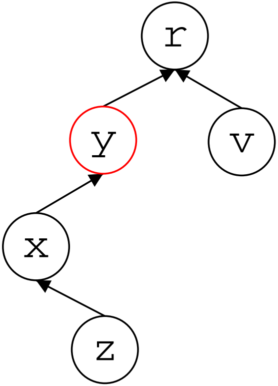
`find(x) =`



Tree graphics borrowed from Tom Schrijvers by kind permission.

Basic Union-Find

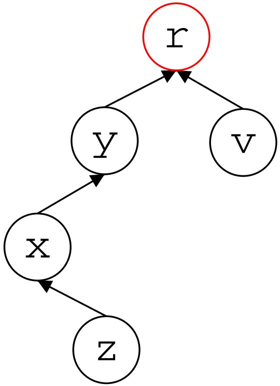
find(x) =



Tree graphics borrowed from Tom Schrijvers by kind permission.

Basic Union-Find

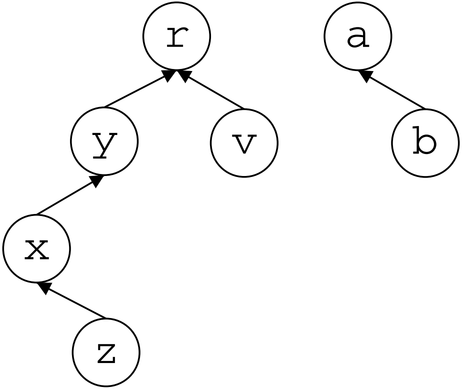
`find(x) = r`



Tree graphics borrowed from Tom Schrijvers by kind permission.

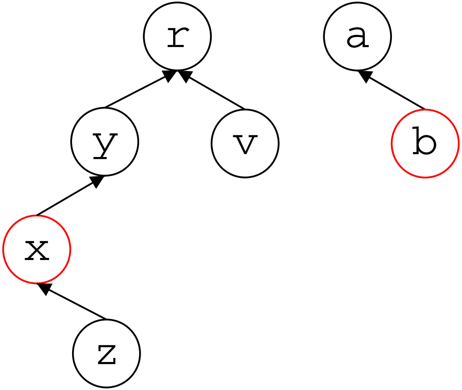
Basic Union-Find

union(x,b):



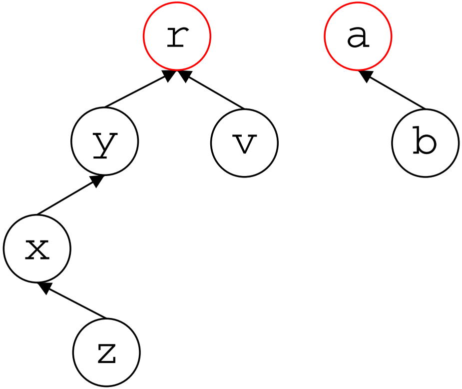
Basic Union-Find

union(x,b): find(x),find(b)



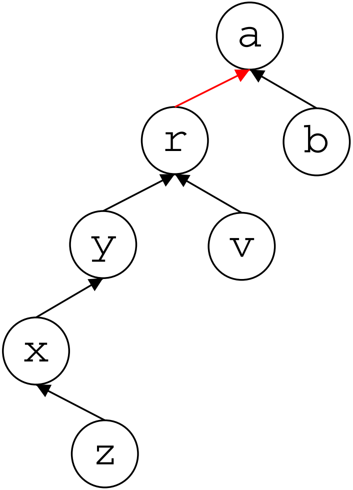
Basic Union-Find

union(x,b): find(x),find(b)



Basic Union-Find

`union`(x,b): `link`(`find`(x),`find`(b))



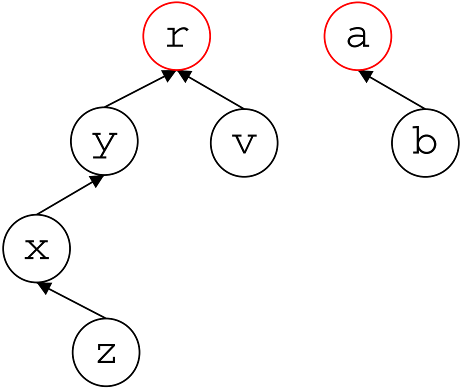
Optimal Union-Find

[Tarjan+, JACM 31(2)]
Optimizations:
Path compression for `find`: point nodes on find path directly to the root.
Union-by-rank for `link`: point root of smaller tree to higher tree.

Logarithmic worst-case time complexity per operation.
Amortized quasi-constant time complexity per operation.

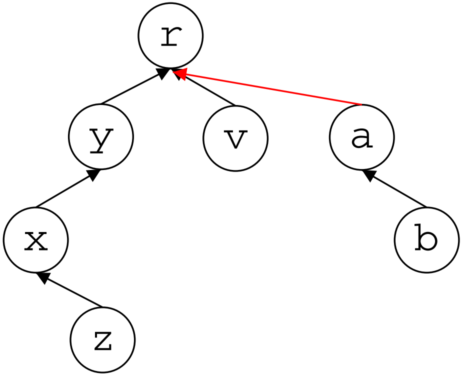
Union-by-rank

[Tarjan+, JACM 31(2)]
Union-by-rank for `link`: point root of smaller tree to higher tree.
rank[r] $= 3$
rank[a] $= 1$



Union-by-rank

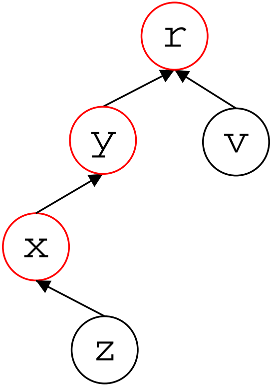
[Tarjan+, JACM 31(2)]
Union-by-rank for `link`: point root of smaller tree to higher tree.
rank[r] = 3
rank[a] = 1



Path Compression for find

[Tarjan+, JACM 31(2)]

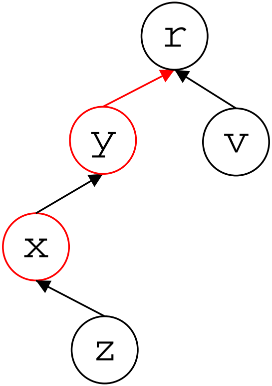
Path compression for `find`: point nodes on find path directly to the root.



Path Compression for find

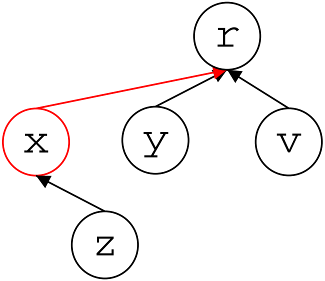
[Tarjan+, JACM 31(2)]

Path compression for find: point nodes on find path directly to the root.



Path Compression for find

[Tarjan+, JACM 31(2)]
Path compression for `find`: point nodes on find path directly to the root.



Basic Union-Find in CHR

Schrijvers/Frühwirth, TPLP Journal Programming Pearl, 2006.

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


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

findRoot @ root(X) \ find(X,R) <=> R=X.

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

Optimal Union-Find in CHR

Schrijvers/Frühwirth, TPLP Journal Programming Pearl, 2006.

```
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) \ find(X,R) <=> 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)).
```

Properties of Union-Find in CHR

Union-Find Algorithm

- Quasi-linear amortised time and space complexity
- Compute most general solution
  - finds relation between given variables
  - checks implication/entailment
  - normalizes solution

Constraint Handling Rules

- Anytime and online algorithm
  - partial solution between rule applications
  - incremental, one-by-one processing
  - variable-disjoint parts in parallel (but not confluent)

Well-suited for constraint solvers.

Optimal Union-Find in CHR

Schrijvers/Frühwirth, TPLP Journal Programming Pearl, 2006.

```
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) \ find(X,R) <=> 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)).
```

Generalised Union-Find in CHR

Introduce arbitrary binary relations between nodes.

```
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,_) \ find(X,XR,R) <=> 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)).
```

Operations on Relations

Operations on relations:

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* $r_1 = id$

Combination of four relations according to commutative diagram:

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

```
X -- XA* -- A
|           |
XY         AB?
|           |
Y -- YB* -- B
```

Instance Boolean Equations

Relations are `eq` and `ne`, truth values are `0` and `1`.

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

```
?- 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.
```

Related Work.
Special case of 2-SAT [Aspvall/Plass/Tarjan 1978] (but not Horn-SAT).
Satisfiability check in linear time only if relations are in specific order.
Uses maximal strongly connected graphs and value propagation.
Result less informative about relations between variables.

Instance Linear Polynomials

Infinite number of relations over infinite domain.
`X-A|B->Y` means `X=A*Y+B` where $A \neq 0$.

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

```
?- 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.
```

Code will fail if variable is fixed, e.g. `link(X,2|1,X)` (`X=-1`).

Instance Linear Polynomials II

Special `linkEq` rules for fixed variables and numeric values.

```
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).
```

```
?- 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.
```

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

```
root(1,9), X=5, Y=1.
```

Related Work. Similar to [Aspvall/Shiloach 1980].
Solves in linear time only if relations are in specific order.
Uses maximal strongly connected graphs and spanning trees.
Result less informative about relations between variables.

Complexity

If we specialise our algorithm in the case where the only relation is `id`, we get back the original program.

Same quasi-linear time and space complexity as the original union-find algorithm if the operations on relation take constant time and space.

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

Mapping function removes the additional arguments and additional built-in constraints.

Use induction on length of derivation and case analysis of the rules applicable in a derivation step.

Correctness

The logical reading of the rules is a consequence of a theory for the relations if these relations are bijective functions.

Proof. Replace `union`, `find`, `link` and `->` by their relations.

```
(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)))
```

Correctness

The logical reading of the rules is a consequence of a theory for the relations if these relations are bijective functions.
Proof. Replace `union`, `find`, `link` and `->` by their relations.

Logical reading of rule `findNode` restricts the allowed relations.

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

```
              Y                        Y
             /                          \
            /                            \
          XY            <=>               YR
          /                                \
         /                                  \
        X---- XR ---R              X---- XR ---R
```

E.g. does not hold for $\leq$=`XR=YR=XY` even though $\leq \circ \leq = \leq$.
Holds for bijective functions, since one fixed variable fixes all the others.

Conclusion

Work in Progress

Simple generalisation of union-find from equality to bijective functions.

- equality and inequality over Booleans
- linear polynomial equations in two variables.

Well-suited for constraint solvers.

Good properties of union-find in CHR are kept.

- quasi-linear time and space efficiency
- most general normalised solution
- checks implication/entailment
- anytime and online parallelisable algorithm

Future Work

- More relations than bijective functions
- Relationship with classes of tractable constraints
- Tradeoff between efficiency and precision

Acknowledgements. Tree graphics from Tom Schrijvers by kind permission.