Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis

Thom Frühwirth

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

ICLP 2005, Barcelona, October 2005

Motivation

Classical optimal union-find algorithm [Tarjan+, JACM 31(2)] implementable in CHR with optimal time complexity [Schrijvers+, WCLP'05,TPLP].

Parallel implementation? Hard problem:

- No parallel computation model for CHR.
- Optimal union-find hard to parallelize.
- Parallel code close to sequential one.

Semi-automatic confluence analysis of sequential program helps to derive parallel program.

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
- All algorithms implementable with optimal time and space complexity
- Incrementality and concurrency for free (on-line, any-time)
- 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

Operational Semantics

Apply rules until exhaustion in any order (fixpoint computation).

Simplify

If	$(H \Leftrightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and	$CT \models G_{builtin} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then	$H' \wedge G \mapsto G \wedge H{=}H' \wedge B$

Propagate

If	$(H \Rightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and	$CT \models G_{builtin} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then	$H' \wedge G \mapsto H' \wedge G \wedge H{=}H' \wedge B$

Refined operational semantics [Duck+, ICLP 2004]: Similar to Prolog, CHR constraints evaluated depth-first from left to right and rules applied top-down in program text order.

Operational Semantics

Apply rules until exhaustion in any order (fixpoint computation).

Simplify

If $(H \Leftrightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and $CT \models G_{builtin} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then $H' \wedge G \mapsto G \wedge H{=}H' \wedge B$

Propagate

If $(H \Rightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and $CT \models G_{builtin} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then $H' \wedge G \mapsto H' \wedge G \wedge H{=}H' \wedge B$

Refined operational semantics [Duck+, ICLP 2004]: Similar to Prolog, CHR constraints evaluated depth-first from left to right and rules applied top-down in program text order.

Operational Semantics

Apply rules until exhaustion in any order (fixpoint computation).

Simplify

If	$(H \Leftrightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and	$CT \models G_{builtin} \rightarrow \exists\bar{x}(H{=}H' \wedge C)$
then	$H' \wedge G \mapsto G \wedge H{=}H' \wedge B$

Propagate

If	$(H \Rightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and	$CT \models G_{builtin} \rightarrow \exists\bar{x}(H{=}H' \wedge C)$
then	$H' \wedge G \mapsto H' \wedge G \wedge H{=}H' \wedge B$

Refined operational semantics [Duck+, ICLP 2004]: Similar to Prolog, CHR constraints evaluated depth-first from left to right and rules applied top-down in program text order.

Operational Semantics

Apply rules until exhaustion in any order (fixpoint computation).

Simplify

If	$(H \Leftrightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and	$CT \models G_{builtin} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then	$H' \wedge G \mapsto G \wedge H{=}H' \wedge B$

Propagate

If	$(H \Rightarrow C \mid B)$ rule with renamed fresh variables $\bar{x}$
and	$CT \models G_{builtin} \rightarrow \exists \bar{x}(H{=}H' \wedge C)$
then	$H' \wedge G \mapsto H' \wedge G \wedge H{=}H' \wedge B$

Refined operational semantics [Duck+, ICLP 2004]: Similar to Prolog, CHR constraints evaluated depth-first from left to right and rules applied top-down in program text order.

Parallelism for CHR

Interleaving semantics: Parallel computation step can be simulated by a sequence of sequential computation steps, similar to e.g. [Saraswat+, POPL'90/'91].

Instantaneous rule applications assumed.

Monotonicity of CHR (Theorem)

$$\frac{A \quad \mapsto \quad B}{A \wedge C \quad \mapsto \quad B \wedge C}$$

Applicable rule remains applicable in any larger context.

Parallelism for CHR

Interleaving semantics: Parallel computation step can be simulated by a sequence of sequential computation steps, similar to e.g. [Saraswat+, POPL'90/'91].

Instantaneous rule applications assumed.

Trivial Confluence of CHR (Corollary)

$$\frac{\begin{array}{lll} A & \mapsto & B \\ C & \mapsto & D \end{array}}{\begin{array}{lll} A \wedge C & \mapsto S \mapsto & B \wedge D \\ (S \text{ either} & A \wedge D & \text{or } B \wedge C) \end{array}}$$

Rule applications on different parts of goal can be exchanged.
Rule applications from different goals can be composed.

Parallelism for CHR

Interleaving semantics: Parallel computation step can be simulated by a sequence of sequential computation steps, similar to e.g. [Saraswat+, POPL'90/'91].

Instantaneous rule applications assumed.

Weak Parallelism of CHR (Definition)

$$\frac{\begin{array}{lcl} A & \mapsto & B \\ C & \mapsto & D \end{array}}{\begin{array}{lcl} A \wedge C & \mapsto & B \wedge D \end{array}}$$

Parallel rule applications to different parts.
Parallel short-cut justified by trivial confluence.

Parallelism for CHR

Interleaving semantics: Parallel computation step can be simulated by a sequence of sequential computation steps, similar to e.g. [Saraswat+, POPL'90/'91].

Instantaneous rule applications assumed.

Strong Parallelism of CHR (Corollary)

$$\frac{\begin{array}{lcl} A \wedge E & \mapsto & B \wedge E \\ C \wedge E & \mapsto & D \wedge E \end{array}}{A \wedge E \wedge C \quad \mapsto \quad B \wedge E \wedge D}$$

Parallel rule applications to overlapping parts, if overlap kept.
Derived from weak parallelism and monotonicity.

Basic Union-Find in CHR

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

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

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

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

Tree data constraints: `root/1` and `~>/2` (points to).
Allowed query: sequence of `make`, `union` operations.

Basic Union-Find in CHR

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

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

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

```
make(a), make(b), union(a,b), find(b,X).
| make
root(a), make(b), union(a,b), find(b,X).
          | make
root(a), root(b), union(a,b), find(b,X).
                   | union
root(a), root(b), find(a,U),  find(b,V), link(U,V), find(b,X).
```

Basic Union-Find in CHR

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

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

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

```
root(a), root(b), find(a,U),  find(b,V), link(U,V), find(b,X).
                  | findRoot
root(a), root(b), U=a,        find(b,V), link(U,V), find(b,X).
                              | findRoot
root(a), root(b), U=a,        V=b,       link(U,V), find(b,X).
                                         | link
root(a),          U=a,        V=b,       V ~> U,    find(b,X).
```

Basic Union-Find in CHR

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

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

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

```
root(a),        U=a,      V=b,      V ~> U,    find(b,X).
                                               | findNode
root(a),        U=a,      V=b,      V ~> U,    find(a,X).
                                               | findRoot
root(a),        U=a,      V=b,      V ~> U,    X=a.
```

Confluence for Parallelism

Maximize independence, non-intereference of rule applications.
Avoid waiting and deadlocks. Robust against local failures.

```
make(a), make(b), union(a,b), find(b,X).
| make    | make    | union
root(a), root(b), find(a,U),  find(b,V), link(U,V), find(b,X).
                  | findRoot  | findRoot            | findRoot
root(a), root(b), U=a,        V=b,       link(U,V), X=b.
                                         | link
root(a),          U=a,        V=b,       V ~> U,    X=b.
```

Note that X=b, it was X=a before.
Result depends on order of rule applications.

Confluence

Result of a query is always the same, no matter which of the applicable rules are applied.

$$
\begin{array}{ccc}
A & \mapsto & B \\
A & \mapsto & C \\
\hline
B & \mapsto^* & D \\
C & \mapsto^* & D
\end{array}
$$

⇒ Independence from the order in which constraints processed.
⇒ Consistency of logical reading of the program.
Decidable, sufficient and necessary condition for confluence of terminating CHR programs through joinability of critical pairs.

Critical pair results from applying two rules to an overlap.
Overlap takes both rule heads and guards, shares some CHR constraints.
Confluence checker available.

Non-Confluence of Basic Sequential Union-Find

Found 8 non-joinable critical pairs [Schrijvers+, WCLP'05].

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

Incompatible tree constraints, trivial (3), e.g.

```
            root(A),A~>B,find(A,X)
   findRoot /                    \ findNode
root(A),A~>B,X=A          root(A),A~>B,find(B,X)
```

Not possible for allowed queries.

Non-Confluence of Basic Sequential Union-Find

Found 8 non-joinable critical pairs [Schrijvers+, WCLP'05].

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

Pending competing links, non-trivial (4), e.g.

```
        root(A),root(B),link(B,A),link(A,B)
      link /                          \ link
root(B),A~>B,link(A,B)        root(A),link(B,A),B~>A
```

Deadlock for linking possible.

Non-Confluence of Basic Sequential Union-Find

Found 8 non-joinable critical pairs [Schrijvers+, WCLP'05].

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

Relative order of find and link, non-trivial (1)

```
            find(B,A),root(B),root(C),link(C,B)
    findRoot /                               \ link
       link /                                 \ findNode
           /                                   \ findRoot
root(C),B~>C,A=B                       root(C),B~>C,A=C
```

Find wrongly returns interior node instead of root.

Trivial Non-Confluence

Same non-joinable critical pairs are less cumbersome.
Combinatorial explosion in the number of critical pairs with program size.

$\Rightarrow$ Ignore "trivial" non-joinable critical pairs:

- not possible for allowed queries: incompatible tree constraints.
- results that are considered equivalent: tree constraints that describe the same sets are equivalent.
- not possible with refined semantics, but with parallel: competing link operations, different orders for find and link.

Insight #1 - Replace = by found

Replace built-in by CHR constraint.

```
            find(B,A),root(B),root(C),link(C,B)
    findRoot /                              \ link
       link /                                \ findNode
           /                                  \ findRoot
root(C),B~>C,A=B                        root(C),B~>C,A=C
```

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

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

Insight #1 - Replace = by found

Replace built-in by CHR constraint.

```
             find(B,A),root(B),root(C),link(C,B)
    findRoot /                                  \ link
       link /                                    \ findNode
           /                                      \ findRoot
root(C),B~>C,found(B,A)                 root(C),B~>C,found(C,A)
```

```
findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot1 @ root(A) \ find(A,X) <=> found(A,X).

linkEq1   @ link(X,Y),found(A,X),found(A,Y) <=> true.
link1     @ link(X,Y),found(A,X),found(B,Y),root(A),root(B) <=>
                                                 B ~> A, root(A).
```

Insight #1 - Replace = by found

Replace built-in by CHR constraint.

```
             find(B,A),root(B),root(C),link(C,B)
    findRoot /                                 \ link
       link /                                   \ findNode
     found /                                     \ findRoot
root(C),B~>C,found(C,A)               root(C),B~>C,found(C,A)
```

```
found     @ A ~> B \ found(A,X) <=> found(B,X).

findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot1 @ root(A) \ find(A,X) <=> found(A,X).

linkEq1   @ link(X,Y),found(A,X),found(A,Y) <=> true.
link1     @ link(X,Y),found(A,X),found(B,Y),root(A),root(B) <=>
                                              B ~> A, root(A).
```

Practically Confluent! 89 non-joinable critical pairs, 84 between link rules, all are trivial. ⇒ Parallel version with just one more rule!

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. Rank.

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

Parallelization problem:
Mingled tree updates result in high, even cyclic trees, undo optimizations.
- Restrict parallelism, auxiliary data/operations [Anderson+, STOC'91]:
  Counters for nodes or time-stamps, or using path halving compression.
- Use other algorithm [Atallah+, JACM 41(6)].

Our solution: Explicit compression after linking to node used for linking.
Correct and deadlock-free.

Union-Find in CHR

Basic union-find

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

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

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

Union-Find in CHR

Basic union-find parallelized

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

findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot1 @ root(A) \ find(A,X) <=> found(A,X).

linkEq1   @ link(X,Y),found(A,X),found(A,Y) <=> true.
link1     @ link(X,Y),found(A,X),found(B,Y),root(A),root(B) <=>
                                                B ~> A, root(A).

found     @ A ~> B \ found(A,X) <=> found(B,X).
```

Union-Find in CHR

Optimal union-find [Schrijvers+, WCLP'05,TPLP]

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

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

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

Union-Find in CHR

Optimal union-find parallelized according to basic version

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

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

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

found     @ A ~> B \ found(A,X) <=> found(B,X).
```

Confluence

For sequential optimal union-find program, confluence checker finds 73 non-joinable critical pairs, 68 of the link rules, analogous to the ones for the basic program.

Only one non-trivial critical pair for competing `find` operations:

```
              find(A,X),A~>B,find(A,Y)
        findNode /            \ findNode
       findNode /              \ findNode
find(X,Y),find(B,X),A~>Y    find(Y,X),find(B,Y),A~>X
```

Parallel finds must wait for result of compression from one of the finds.

Insight #2 - Explicit Compression

Keep tree constraint, postpone compression until root.
All finds can proceed, first one finished can compress.

```
findNode1 @ A~>B \ find(A,X) <=> find(B,X), compr(A,X).
found1    @ A~>B \ found(A,X) <=> found(B,X), compr(A,X).
compress1 @ root(C,N),found(C,X) \ A~>B,compr(A,X) <=> A~>C.
```

Compression only immediately before linking, deadlocks after linking.
Keeping `found` would be too expensive because of rule `found1`.

Insight #3 - Compress to Linking Root

Compress the nodes of a path to the root used for linking.
$\Rightarrow$ Compression is performed after linking.
Implementation: Linking replaces `found` by new `foundc`.

```
linkEq2   @ link(X,Y),found(A,X), found(A,Y) <=>
                      foundc(A,X),foundc(A,Y).
linkLeft2 @ link(X,Y),found(A,X),found(B,Y),root(A,N),root(B,M)
               N>=M | foundc(A,X),foundc(B,Y),
                      B ~> A, N1 is max(N,M+1), root(A,N1).
```

Compression cannot deadlock anymore.

Optimal Union-Find Parallelized

```
findNode1 @ A ~> B \ find(A,X) <=> find(B,X), compr(A,X).
findRoot1 @ root(A,N) \ find(A,X) <=> found(A,X).

found1    @ A ~> B \ found(A,X) <=> found(B,X), compr(A,X).

compress  @ foundc(C,X) \ A ~> B, compr(A,X) <=> A ~> C.

linkEq2   @ link(X,Y),found(A,X), found(A,Y) <=>
                      foundc(A,X),foundc(A,Y).
linkLeft2 @ link(X,Y),found(A,X),found(B,Y),root(A,N),root(B,M) <=>
                N>=M | foundc(A,X),foundc(B,Y),
                       B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight2@ link(Y,X),found(A,X),found(B,Y),root(A,N),root(B,M) <=>
                N>=M | foundc(A,X),foundc(B,Y),
                       B ~> A, N1 is max(N,M+1), root(A,N1).
```

Confluence of Parallelized Optimal Union-Find

Confluence checker finds 686 non-joinable critical pairs, 671 of the link rules.
35 trivial critical pairs due to equivalence of the nodes in the trees.

Only one non-trivial critical pair for competing compressions:

```
compress   @ foundc(C,X) \ A ~> B, compr(A,X) <=> A ~> C.
```

```
      foundc(C,X),compr(A,X), A~>B, foundc(D,Y),compr(A,Y)
    compress /                                \ compress
   compress /                                  \ compress
foundc(C,X),foundc(D,Y), A~>D     foundc(C,X),foundc(D,Y), A~>C
```

Different trees denote same set, as nodes `A, B, C, D` on the same path.

Correctness of Union-Find in CHR

Optimal sequential union-find proven correct under refined semantics [Schrijvers+, TPLP].

- Show that parallel and sequential programs simulate each other.
  Map computations between them:
  Map states (constraints) and computation steps (rule applications).
- Mapping based on transformation from sequential to parallel program:
  Built-in = replaced by CHR constraints.
  `found` behaves like `find` until `link`,
  then replaced by `foundc` that is like = and
  used by `compr` for explicit compression.

⇒ Parallel program simulates sequential program.
⇒ Sequential program simulates parallel program for non-competing compressions.

Sequential to Parallel

Result of find and path compression are made explicit.

- Built-in = produced by `findRoot` replaced by `found` constraints until involved in linking. From then on, replaced by `foundc`.
- Immediately after linking, insert applications of the `compress` rule into the resulting parallel computation, so that compression is actually performed (removing all `compr` constraints).

$\Rightarrow$ Parallel program simulates sequential program.

Parallel to Sequential

Interleaving semantics gives sequentialized parallel computation.

- Map `found` into `find` and thus rule `found1` into `findNode`. So rule `findRoot1` rule becomes obsolete, because it does not change any constraints under the mapping.
- Just before linking, insert two `findRoot` rule applications for the two involved `find`.
- After linking, map `foundc` constraints into `=`.
- Map constraints `A~>B,compr(A,X)` into `A~>X`. So `compress` rule becomes obsolete.

Parallel to Sequential

Interleaving semantics gives sequentialized parallel computation.

- Map `found` into `find` and thus rule `found1` into `findNode`. So rule `findRoot1` rule becomes obsolete, because it does not change any constraints under the mapping.
- Just before linking, insert two `findRoot` rule applications for the two involved `find`.
- After linking, map `foundc` constraints into `=`.
- Map constraints `A~>B,compr(A,X)` into `A~>X`. So `compress` rule becomes obsolete.

But consider critical pair for competing compressions:

```
      foundc(C,X),compr(A,X), A~>B, foundc(D,Y),compr(A,Y)
    compress /                                        \ compress
   compress /                                          \ compress
foundc(C,X),foundc(D,Y),A~>D       foundc(C,X),foundc(D,Y),A~>C
```

Only one of the computations simulated by the sequential program.

Parallel to Sequential

Interleaving semantics gives sequentialized parallel computation.

- Map `found` into `find` and thus rule `found1` into `findNode`. So rule `findRoot1` rule becomes obsolete, because it does not change any constraints under the mapping.
- Just before linking, insert two `findRoot` rule applications for the two involved `find`.
- After linking, map `foundc` constraints into `=`.
- Map constraints `A~>B,compr(A,X)` into `A~>X`. So `compress` rule becomes obsolete.

⇒ Sequential program simulates parallel program for non-competing compressions.

Conclusion

Exploratory Paper

- Parallel execution model for CHR.
- Parallelized basic and optimal sequential classical union-find algorithm in CHR using confluence analysis and three insights.
- Code is close to the original, promises to be as efficient:
  - Find operations can run in parallel but not competing linking.
  - Additional parallel work for `found` and `compr`.

Future Work

- more parallel union-find correctness and time complexity analysis,
- more parallel CHR model, its implementation and empirical evaluation,
- more practical confluence analysis: triviality of critical pairs,
- confluence-based parallelization methodology,
  apply to more CHR programs, in particular constraint solvers,
  can also transform refined into standard semantics programs.

Acknowledgements Marc Meister, Tom Schrijvers, referees.