Guard Simplification in CHR programs

WCLP 2005
Ulm, Germany

Jon Sneyers, Tom Schrijvers, Bart Demoen

{jon,toms,bmd}@cs.kuleuven.ac.be.

K.U.Leuven, Belgium
Overview

1. Constraint Handling Rules (CHR)
2. $\omega_t$ and $\omega_r$ semantics
3. Guard simplification
   - 3.1 Basic idea
   - 3.2 Head matching simplification
   - 3.3 Type and mode declarations
4. Implementation
5. Experimental results
6. Conclusion & Future work
Write your own constraint solver as CHRs:
- application tailored solvers
- embedded in Prolog (or other host language) $\Rightarrow$ no interfacing problems
- high-level specification
  - focus on what, not how
  - fixpoint computation is taken care of
  - very compact programs
  - easy to understand, modify and experiment with

Also useful as a general-purpose language
1. Constraint Handling Rules

Three kinds of CHR rules:

- **Simplification rules:**
  
  RemovedHeads $\leftrightarrow$ Guard $\mid$ Body.

- **Propagation rules:**
  
  KeptHeads $\Rightarrow$ Guard $\mid$ Body.

- **Simpagation rules:**
  
  KeptHeads $\setminus$ RemovedHeads $\leftrightarrow$ Guard $\mid$ Body.
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.
Constraint store:
V in 3:8
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.

Constraint store:
V in 3:8
V in 2:6
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.

Constraint store:
V in 3:8
V in 2:6
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.
Constraint store:
V in 3:6
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.

Constraint store:
V in 3:6
V in 0:3
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.

Constraint store:
V in 3:6
V in 0:3
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.

Constraint store:
V in 3:3
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.  

Constraint store:
V in 3:3
1. CHR: Example: interval solver

:- constraints in/2.
X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

Prompt:
?- V in 3:8, V in 2:6, V in 0:3.

V = 3
Yes
2. Operational semantics

- theoretical operational semantics $\omega_t$
  - rules can be applied in any order
  - confluence is nontrivial

- refined operational semantics $\omega_r$
  - an instance of $\omega_t$
  - rules applied in textual order
  - better termination/complexity possible
  - used in all major CHR implementations
Most CHR programmers use the refined operational semantics ($\omega_r$).

<table>
<thead>
<tr>
<th>Feature</th>
<th>$\omega_t$</th>
<th>$\omega_r$</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear logical reading</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>easy to implement things like key lookup</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>efficient compiled code</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>programs are correct under other semantics</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>
3.1 Guard Simplification: Basic Idea

X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

- When last rule is tried, first two rules did not fire
  (otherwise active constraint was removed)
  ⇒ guards of first two rules failed for both constraints
3.1 Guard Simplification: Basic Idea

X in A:B <=> A>B | fail.
X in A:B <=> A =:= B | X is A.

■ When last rule is tried, first two rules did not fire
  (otherwise active constraint was removed)
  ⇒ guards of first two rules failed for both constraints

■ negations of first guard: A =< B and C =< D
  negations of second guard: A =\= B and C =\= D
  ⇒ this entails A < B and C < D
  ⇒ guard of last rule can be simplified
3.1 Guard Simplification: Basic Idea

X in A:B ⇔ A>B | fail.
X in A:B ⇔ A =:= B | X is A.
X in A:B, X in C:D ⇔ X in max(A,C):min(B,D).

- When last rule is tried, first two rules did not fire
  (otherwise active constraint was removed)
  ⇒ guards of first two rules failed for both constraints

- negations of first guard: A =< B and C =< D
  negations of second guard: A =\= B and C =\= D
  ⇒ this entails A < B and C < D
  ⇒ guard of last rule can be simplified
3.1 Guard Simplification: Basic Idea

- Order of rules is implicit precondition
- Programmers don’t write guards entailed by rule order
  - ✔ more compact and efficient programs
  - ✗ logical reading obfuscated
3.1 Guard Simplification: Basic Idea

- Order of rules is implicit precondition
- Programmers don’t write guards entailed by rule order
  - ✓ more compact and efficient programs
  - × logical reading obfuscated
- Solution: Guard Simplification
  - ✷ Automatically remove entailed guards
  - ✷ Programmer can put all preconditions in guard
    ⇒ logical reading restored
- Advantages of both $\omega_t$ and $\omega_r$!
3.1 GS: Basic Idea: in general

\[
\begin{align*}
KH_1^1, KH_2^1, \ldots KH_{k_1}^1 & \setminus RH_1^1, RH_2^1, \ldots RH_{r_1}^1 \iff G^1 | B^1. \\
KH_1^2, KH_2^2, \ldots KH_{k_2}^2 & \setminus RH_1^2, RH_2^2, \ldots RH_{r_2}^2 \iff G^2 | B^2. \\
\vdots & \\
KH_1^i, KH_2^i, \ldots KH_{k_i}^i & \setminus RH_1^i, RH_2^i, \ldots RH_{r_i}^i \iff G^i | B^i. \\
H^i & \\
KH_1^n, KH_2^n, \ldots KH_{k_n}^n & \setminus RH_1^n, RH_2^n, \ldots RH_{r_n}^n \iff G^n | B^n.
\end{align*}
\]

The guard of the \(i\)-th rule can be removed if:

\[
\bigwedge \left\{ \neg G^j \mid (1 \leq j < i) \land (H^j \subseteq H^i) \land (r_j > 0) \right\} \rightarrow G^i
\]

rule \(j\) is an earlier subrule of rule \(i\)
3.2 Head matching simplification

\[ p(X, Y) \iff X \ \backslash=\ Y \mid Body1 \]
\[ p(X, X) \iff Body2 \]

- Head matchings are implicit part of guard
3.2 Head matching simplification

\[ p(X, Y) \iff X \backslash= Y \mid \text{Body1} \]
\[ p(X, Y) \iff X = Y \mid \text{Body2} \]

- Head matchings are implicit part of guard
  ⇒ make them explicit and do guard simpl.
3.2 Head matching simplification

\[ p(X, Y) \iff X \triangleq Y \mid \text{Body1} \]
\[ p(X, Y) \iff \text{Body2} \]

- Head matchings are implicit part of guard
  \( \Rightarrow \) make them explicit and do guard simpl.
- More general head
- Easier to detect \( p/2 \) is a never-stored constraint
3.3 Type and mode declarations

\[
\begin{align*}
\text{sum}([], S) & \iff S = 0. \\
\text{sum}([X|Xs], S) & \iff \text{sum}(Xs, S2), S \text{ is } X+S2.
\end{align*}
\]

- \text{sum/2 is never-stored, but we can't detect it}
3.3 Type and mode declarations

```prolog
:- chr_type list(T) —> [] ; [T | list(T)].
:- constraints sum(+list(int),?int).
sum([],S) <=> S = 0.
sum([X|Xs],S) <=> sum(Xs,S2), S is X+S2.
```

- `sum/2` is never-stored, but we can’t detect it

⇒ introduce type and mode declarations
3.3 Type and mode declarations

:- chr_type list(T) -> [] ; [T | list(T)].
:- constraints sum(+list(int), ?int).
sum([], S) <=> S = 0.
sum([X|Xs], S) <=> sum(Xs, S2), S is X+S2.

- `sum/2` is never-stored, but we can’t detect it
  ⇒ introduce type and mode declarations
  ⇒ head matching can be simplified
  in this case, moved to the body because rhs vars are needed
3.3 Type and mode declarations

:- chr_type list(T) ——> [] ; [T | list(T)].
:- constraints sum(+list(int),?int).
sum([],S) <=> S = 0.
sum(A,S) <=> A=[X|Xs], sum(Xs,S2), S is X+S2.

sum/2 is never-stored, but we can’t detect it
⇒ introduce type and mode declarations
⇒ head matching can be simplified
in this case, moved to the body because rhs vars are needed
⇒ easy to detect never-stored property
4. Implementation

- Implemented in K.U. Leuven CHR compiler
  - both in hProlog+CHR and SWI-Prolog+CHR
  - added support for type defs/decls

- Implemented as new compilation phase:
  - source to source transformation
  - derive information from rule order, types, modes
  - if guard is entailed, remove guard
  - if \( \neg \) guard is entailed, give warning & remove rule

- Entailment check module (written in CHR)
Example CHR program:

\[
\text{filter}([X|\text{In}], P, \text{Out}) \iff 0 \leq X \mod P \mid \\
\qquad \text{Out}=[X|\text{Out1}], \\
\qquad \text{filter}((\text{In}, P, \text{Out1})].
\]

\[
\text{filter}([X|\text{In}], P, \text{Out}) \iff 0 =:= X \mod P \mid \\
\qquad \text{filter}((\text{In}, P, \text{Out})].
\]

\[
\text{filter}([], P, \text{Out}) \iff \text{Out}=[].
\]
4. Implementation: generated code

Without guard simplification:

```prolog
filter(List,P,Out) :- filter(List,P,Out, _ ).

filter(List,P,Out,C) :- % first occurrence
  nonvar(List), List = [X|In], 0 =\= X mod P, !,
  ... % remove from constraint store if needed
  Out = [E|Out1], filter(In,P,Out1).

filter(List,P,Out,C) :- % second occurrence
  nonvar(List), List = [X|In], 0 =:= X mod P, !,
  ... % remove from constraint store if needed
  filter(In,P,Out).
```

Jon Sneyers, Tom Schrijvers, Bart Demoen – *Guard Simplification in CHR programs* – WCLP 2005 – p.15
Without guard simplification: (continued)

\[
\text{filter}(\text{List}, \_ \_ \_ , \text{Out}, \text{C}) :- \quad \% \text{ third occurrence }
\]
\[
\text{List} == [], !,
\]
\[
\ldots \% \text{ remove from constraint store if needed }
\]
\[
\text{Out} = [] .
\]

\%
\[
% \text{ insert into store in case none of the rules matched }
\]
\[
\text{filter}(\text{List}, \text{P}, \text{Out}, \text{C}) :- \\
\ldots \% \text{ insert into constraint store }
\]
4. Implementation: generated code

With guard simplification (and type/mode info):  

\[
\text{filter}(\text{List}, \text{P}, \text{Out}) :- \text{List} = [X|\text{In}], \ 0 \leq X \mod E, \ !,  \\
\quad \text{Out} = [X|\text{Out1}], \ \text{filter}(\text{In}, \text{P}, \text{Out1}).
\]

\[
\text{filter}(\text{List}, \text{P}, \text{Out}) :- \text{List} = [\_|\text{In}], \ !, \ \text{filter}(\text{In}, \text{P}, \text{Out}).
\]

\[
\text{filter}(\_, \_, \text{Out}) :- \text{Out} = [\_].
\]
## 5. Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lang</th>
<th>GS</th>
<th>Mode</th>
<th>Type</th>
<th>Codesize</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum (10000,500)</td>
<td>CHR</td>
<td>—</td>
<td>×</td>
<td>×</td>
<td>4,46</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>—</td>
<td>√</td>
<td>×</td>
<td>3,10</td>
<td>88.9%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>2,6</td>
<td>66.7%</td>
</tr>
<tr>
<td>handwritten Prolog code</td>
<td>2,5</td>
<td></td>
<td></td>
<td></td>
<td>66.1%</td>
<td></td>
</tr>
<tr>
<td>Takeuchi (1000)</td>
<td>CHR</td>
<td>×</td>
<td>×</td>
<td>—</td>
<td>4,50</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>×</td>
<td>√</td>
<td>—</td>
<td>3,17</td>
<td>65.8%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>√</td>
<td>—</td>
<td>—</td>
<td>2,12</td>
<td>61.0%</td>
</tr>
<tr>
<td>handwritten Prolog code</td>
<td>2,12</td>
<td></td>
<td></td>
<td></td>
<td>61.0%</td>
<td></td>
</tr>
<tr>
<td>nrev (30,50000)</td>
<td>CHR</td>
<td>—</td>
<td>×</td>
<td>×</td>
<td>8,92</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>—</td>
<td>√</td>
<td>×</td>
<td>6,20</td>
<td>62.9%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>4,11</td>
<td>23.0%</td>
</tr>
<tr>
<td>handwritten Prolog code</td>
<td>4,7</td>
<td></td>
<td></td>
<td></td>
<td>20.5%</td>
<td></td>
</tr>
</tbody>
</table>
## 5. Experimental results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lang</th>
<th>GS</th>
<th>Mode</th>
<th>Type</th>
<th>Codesize</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>cprimes (100000)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHR</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
<td>✓</td>
<td>14 , 160</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>—</td>
<td>✓</td>
<td>11 , 42</td>
<td>58.1%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>12 , 120</td>
<td>99.4%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>10 , 35</td>
<td>57.2%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>8 , 25</td>
<td>55.8%</td>
</tr>
<tr>
<td>handwritten Prolog code</td>
<td>8 , 23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55.8%</td>
</tr>
<tr>
<td>dfsearch (16,500)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHR</td>
<td>✓</td>
<td>✓</td>
<td>—</td>
<td>✓</td>
<td>5 , 67</td>
<td>100.0%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>—</td>
<td>✓</td>
<td>4 , 16</td>
<td>84.4%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>5 , 66</td>
<td>90.7%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>4 , 15</td>
<td>79.4%</td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>3 , 11</td>
<td>59.5%</td>
</tr>
<tr>
<td>handwritten Prolog code</td>
<td>3 , 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55.8%</td>
</tr>
</tbody>
</table>
5. Experimental results

- Generated code much closer to Prolog
- CHR programmers usually write deterministic parts in Prolog for efficiency
  \[ \Rightarrow \text{can write everything in CHR now} \]
- Constraint not never-stored → only minor improvement (guards are usually cheap)
- Never-stored constraint → big improvement
5. Experimental results

- Performance of guard simplification phase itself: reasonable
- Scales well …

- Compiling K.U.Leuven CHR compiler: (139 rules, 73 constraints) — 3.3 seconds, 1.5 for GS
5. Experimental results

- Performance of guard simplification phase itself: reasonable
- Scales well ... except when there are many rules and few constraints!
  → complexity depends on number of earlier subrules
- Compiling K.U.Leuven CHR compiler:
  (139 rules, 73 constraints) — 3.3 seconds, 1.5 for GS
- Compiling entailment checker:
  (123 rules, 3 constraints) — 2.1 seconds, 1.8 for GS!
## 5. Experimental results

<table>
<thead>
<tr>
<th>program</th>
<th>R</th>
<th>C</th>
<th>R/C</th>
<th>GS</th>
<th>TC</th>
<th>GS/TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>union-find</td>
<td>6</td>
<td>6</td>
<td>1.0</td>
<td>0</td>
<td>40</td>
<td>0.0%</td>
</tr>
<tr>
<td>timed automata</td>
<td>23</td>
<td>17</td>
<td>1.4</td>
<td>50</td>
<td>310</td>
<td>16.1%</td>
</tr>
<tr>
<td>well-founded semantics</td>
<td>43</td>
<td>18</td>
<td>2.4</td>
<td>70</td>
<td>340</td>
<td>20.6%</td>
</tr>
<tr>
<td>finite domain solver</td>
<td>13</td>
<td>6</td>
<td>2.2</td>
<td>80</td>
<td>200</td>
<td>40.0%</td>
</tr>
<tr>
<td>CHR compiler</td>
<td>139</td>
<td>73</td>
<td>1.9</td>
<td>1,540</td>
<td>3,270</td>
<td>47.1%</td>
</tr>
<tr>
<td>boolean solver</td>
<td>78</td>
<td>8</td>
<td>9.8</td>
<td>420</td>
<td>590</td>
<td>71.2%</td>
</tr>
<tr>
<td>(in)finite domain solver</td>
<td>81</td>
<td>9</td>
<td>9.0</td>
<td>950</td>
<td>1,250</td>
<td>76.0%</td>
</tr>
<tr>
<td>entailment checker</td>
<td>123</td>
<td>3</td>
<td>41.0</td>
<td>1,830</td>
<td>2,150</td>
<td>85.1%</td>
</tr>
</tbody>
</table>

GS: Guard simplification phase time (ms)  
R: number of rules  
TC: Total compilation time (ms)  
C: number of constraints
6. Conclusion

- GS allows more declarative CHR programs
- Allows other analyses to detect more cases (e.g. never-stored analysis)
- Type and mode information $\rightarrow$ more simplification
- Generated code is as close to Prolog as possible
  - more efficient
  - no need to write parts in Prolog for efficiency
6. Future work

Current work:

- Improve scalability
- Occurrence subsumption: generalizes symmetry analysis:
  \[ c(X) \backslash c(X) \iff \ldots \]
  \[ c(X,Y), c(Y,X) \iff \text{true} \mid \ldots \]
  \[ c(X,Y,Z), c(Y,Z,X), c(Z,X,Y) \iff (p(X); p(Y)) \mid \ldots \]
6. Future work

Future work:

- Support for declarations of properties of user-defined predicates
- Use info derived for GS in other phases
- Program specialization on constraint calls in bodies based on derived properties of call arguments
Questions?

Related papers (same authors):