Constraint Handling Rules
Recent Research
Analysis, Optimizing compilation, Complexity, Extensions

Jon Sneyers
K.U.Leuven, Belgium

April 2008
Overview

1. Introduction to CHR
   - CHR researchers
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimizing Compilation
   - CHR systems
   - Guard reasoning
   - Memory reuse

3. Complexity
   - Asymptotic complexities
   - Constant factors
   - Other declarative languages

4. Extensions of CHR
   - Negation
   - Aggregates
Overview

1. Introduction to CHR
   - CHR researchers
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimizing Compilation
   - CHR systems
   - Guard reasoning
   - Memory reuse

3. Complexity
   - Asymptotic complexities
   - Constant factors
   - Other declarative languages

4. Extensions of CHR
   - Negation
   - Aggregates
High-level language extension

Multi-headed committed-choice guarded rules

Originally designed for constraint solvers

General-purpose programming language

Every algorithm can be implemented with the optimal time and space complexity! [Sneyers-Schrijvers-Demoen CHR’05]
Constraint Handling Rules [Frühwirth 1991]

- High-level language extension
- Multi-headed committed-choice guarded rules
- Originally designed for constraint solvers
- General-purpose programming language
- Every algorithm can be implemented with the optimal time and space complexity! [Sneyers-Schrijvers-Demoen CHR’05]
Main CHR research groups

1. Ulm, Germany    2. Leuven, Belgium    3. Melbourne, Australia
Introduction to CHR
Analysis and Optimizing
Syntax, semantics, results
Examples

CHR researchers

Jon Sneyers
Constraint Handling Rules: recent research
The Leuven CHR team

- Tom Schrijvers
  (since 2002)
- Bart Demoen
  (since 2002)
- Jon Sneyers
  (since summer 2004)
- Leslie De Koninck
  (since summer 2005)
- Peter Van Weert
  (since early 2006)
- Paolo Pilozzi
  (since summer 2006)
- Dean Voets
  (since summer 2007)
- Pieter Wuille
  (since summer 2007)

- INCLP(R)
- CHR(Java)
- termination
- CHR(C)
The Leuven CHR team

- Tom Schrijvers (since 2002)
- Bart Demoen (since 2002)
- Jon Sneyers (since summer 2004)
- Leslie De Koninck (since summer 2005)
- INCLP(R)
- Peter Van Weert (since early 2006)
- CHR(Java)
- Paolo Pilozzi (since summer 2006)
- termination
- Dean Voets (since summer 2007)
- termination
- Pieter Wuille (since summer 2007)
- CHR(C)
The Leuven CHR team

- Tom Schrijvers (since 2002)
- Bart Demoen (since 2002)
- Jon Sneyers (since summer 2004)
- Leslie De Koninck (since summer 2005)
- Peter Van Weert (since early 2006)
- Paolo Pilozzi (since summer 2006)
- Dean Voets (since summer 2007)
- Pieter Wuille (since summer 2007)

Projects:
- INCLP(R)
- CHR(Java)
- CHR(C)
- termination
- termination
The Leuven CHR team

- Tom Schrijvers (since 2002)
- Bart Demoen (since 2002)
- Jon Sneyers (since summer 2004)
- Leslie De Koninck (since summer 2005) INCLP(R)
- Peter Van Weert (since early 2006) CHR(Java)
- Paolo Pilozzi (since summer 2006) termination
- Dean Voets (since summer 2007) termination
- Pieter Wuille (since summer 2007) CHR(C)
The Leuven CHR team

Tom Schrijvers (since 2002)

Bart Demoen (since 2002)

Jon Sneyers (since summer 2004)

Leslie De Koninck (since summer 2005) INCLP(R)

Peter Van Weert (since early 2006) CHR(Java)

Paolo Pilozzi (since summer 2006) termination

Dean Voets (since summer 2007) termination

Pieter Wuille (since summer 2007) CHR(C)

Constraint Handling Rules: recent research
The Leuven CHR team

- Tom Schrijvers  
  (since 2002)
- Bart Demoen  
  (since 2002)
- Jon Sneyers  
  (since summer 2004)
- Leslie De Koninck  
  (since summer 2005)  
  \textit{INCLP(R)}
- Peter Van Weert  
  (since early 2006)  
  \textit{CHR(Java)}
- Paolo Pilozzi  
  (since summer 2006)  
  \textit{termination}
- Dean Voets  
  (since summer 2007)  
  \textit{termination}
- Pieter Wuille  
  (since summer 2007)  
  \textit{CHR(C)}
Syntax and semantics of CHR on 1 slide

- **CHR(\(X\))** where \(X\) is host-language
  - CHR constraints, defined in CHR program
  - Built-in (host-language) constraints, theory \(CT\)

**Syntax:** CHR program consists of rules

- Simplification: \(h \iff g \mid b\)
- Propagation: \(h \rightarrow g \mid b\)
- Simpagation: \(h_1 \setminus h_2 \iff g \mid b\)
- head \(h\), guard \(g\) and body \(b\) are conjunctions of constraints
  (\(h\): only CHR constraints; \(g\): only host-language constraints)

**Logical semantics:** rules define theory \(P\)

- Simplification: \(g \rightarrow (h \leftrightarrow b)\)
- Propagation: \(g \rightarrow (h \rightarrow b)\)
- Simpagation: \(g \rightarrow (h_1 \rightarrow (h_2 \leftrightarrow b))\)

**Operational semantics:** rules manipulate constraint store

- if \(h\) is in constraint store and \(g\) holds, then add \(b\)
- Simplification: remove \(h\);
- Propagation: keep \(h\);
- Simpagation: keep \(h_1\), remove \(h_2\)
CHR(X) where X is host-language
- CHR constraints, defined in CHR program
- Built-in (host-language) constraints, theory CT

Syntax: CHR program consists of rules
- Simplification: \( h \iff g \!\!\mid b \)
- Propagation: \( h \rightarrow g \!\!\mid b \)
- Simpagation: \( h_1 \backslash h_2 \iff g \!\!\mid b \)
- head \( h \), guard \( g \) and body \( b \) are conjunctions of constraints
  (\( h \): only CHR constraints; \( g \): only host-language constraints)

Logical semantics: rules define theory \( \mathcal{P} \)
- Simplification: \( g \rightarrow (h \leftrightarrow b) \)
- Propagation: \( g \rightarrow (h \rightarrow b) \)
- Simpagation: \( g \rightarrow (h_1 \rightarrow (h_2 \leftrightarrow b)) \)

Operational semantics: rules manipulate constraint store
- if \( h \) is in constraint store and \( g \) holds, then add \( b \)
- Simplification: remove \( h \); Propagation: keep \( h \);
  Simpagation: keep \( h_1 \), remove \( h_2 \)
CHR(\(X\)) where \(X\) is host-language e.g. \(X = \text{Prolog}\)

- CHR constraints, defined in CHR program e.g. \(<\)
- Built-in (host-language) constraints, theory \(CT\) e.g. \(=\)

**Syntax:** CHR program consists of rules

- Simplification: \(h \iff g | b\) e.g. \(A \leq B, B \leq A \iff A = B\).
- Propagation: \(h \rightarrow g | b\) e.g. \(A \leq B, B \leq C \rightarrow A \leq C\).
- Simpagation: \(h_1 \setminus h_2 \iff g | b\) e.g. \(A \leq B \setminus A \leq B \iff \text{true}\).
- head \(h\), guard \(g\) and body \(b\) are conjunctions of constraints
  (\(h\): only CHR constraints; \(g\): only host-language constraints)

**Logical semantics:** rules define theory \(\mathcal{P}\)

- Simplification: \(g \rightarrow (h \leftrightarrow b)\)
- Propagation: \(g \rightarrow (h \rightarrow b)\)
- Simpagation: \(g \rightarrow (h_1 \rightarrow (h_2 \leftrightarrow b))\)

**Operational semantics:** rules manipulate constraint store

- if \(h\) is in constraint store and \(g\) holds, then add \(b\)
- Simplification: remove \(h\); Propagation: keep \(h\);
  Simpagation: keep \(h_1\), remove \(h_2\)
CHR($X$) where $X$ is host-language e.g. $X = \text{Prolog}$

- CHR constraints, defined in CHR program e.g. $\leq$
- Built-in (host-language) constraints, theory $CT$ e.g. $=$

- **Syntax:** CHR program consists of rules
  - Simplification: $h \leftrightarrow g \mid b$ e.g. $A \leq B, B \leq A \iff A = B.$
  - Propagation: $h \implies g \mid b$ e.g. $A \leq B, B \leq C \implies A \leq C.$
  - Simpagation: $h_1 \setminus h_2 \leftrightarrow g \mid b$ e.g. $A \leq B \setminus A \leq B \iff \text{true}.$

  head $h$, guard $g$ and body $b$ are conjunctions of constraints
  ($h$: only CHR constraints; $g$: only host-language constraints)

- **Logical semantics:** rules define theory $\mathcal{P}$
  - Simplification: $g \rightarrow (h \leftarrow b)$
  - Propagation: $g \rightarrow (h \rightarrow b)$
  - Simpagation: $g \rightarrow (h_1 \rightarrow (h_2 \leftarrow b))$

- **Operational semantics:** rules manipulate constraint store
  - if $h$ is in constraint store and $g$ holds, then add $b$
  - Simplification: remove $h$; Propagation: keep $h$
  - Simpagation: keep $h_1$, remove $h_2$
**Syntax and semantics of CHR on 1 slide**

- **CHR**($X$) where $X$ is host-language e.g. $X = $ Prolog
  - CHR constraints, defined in CHR program e.g. $\leq$
  - Built-in (host-language) constraints, theory $CT$ e.g. $=$

- **Syntax**: CHR program consists of rules
  - Simplification: $h \iff g | b$ e.g. $A \leq B, B \leq A \iff A = B$.  
  - Propagation: $h \implies g | b$ e.g. $A \leq B, B \leq C \implies A \leq C$.  
  - Simpagation: $h_1 \setminus h_2 \iff g | b$ e.g. $A \leq B \setminus A \leq B \iff \text{true}$.
  - head $h$, guard $g$ and body $b$ are conjunctions of constraints ($h$: only CHR constraints; $g$: only host-language constraints)

- **Logical semantics**: rules define theory $\mathcal{P}$
  - Simplification: $g \to (h \iff b)$
  - Propagation: $g \to (h \to b)$
  - Simpagation: $g \to (h_1 \to (h_2 \iff b))$

- **Operational semantics**: rules manipulate constraint store
  - if $h$ is in constraint store and $g$ holds, then add $b$
  - Simplification: remove $h$; Propagation: keep $h$; Simpagation: keep $h_1$, remove $h_2$
Program $P$ has a *computation* of a goal $G$ with answer $A$ if initializing the constraint store with $G$ and applying the rules until no more rules are applicable results in $A$.

*Termination*: no infinite computation

*Soundness*. If goal $G$ has a computation with answer $A$, then $P, CT \models G \leftrightarrow A$

*Completeness*. If goal $G$ terminates and $P, CT \models G \leftrightarrow A$, then $G$ has an answer $A'$ such that $P, CT \models A \leftrightarrow A'$,

*Confluence*: if multiple rules are applicable, it does not matter which one is applied — in the end, you get the same answer.

Decidable **confluence test** for terminating programs: “all critical pairs are joinable $\iff$ confluence”

*Correctness*: if $P$ is confluent, then $P \cup CT$ is consistent.
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X<=X <=> true.
antisymmetry @ X<=Y, Y<=X <=> X=Y.
idempotence @ X<=Y \ X<=Y <=> true.
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

Example execution:

- Goal: A<=B, B<=C, C<=A
- Store:
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.

reflexivity @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: A <= B, B <= C, C <= A
- Store: A <= B
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X \= X <=> true.
antisymmetry @ X \= Y, Y \= X <=> X = Y.
idempotence @ X \= Y \\ X \= Y <=> true.
transitivity @ X \= Y, Y \= Z ==> X \= Z.
```

Example execution:

- Goal: A \= B, B \= C, C \= A
- Store: A \= B, B \= C
Example 1: less-or-equal solver

- Typical “solver” CHR program:

  ```prolog
  :- chr_constraint <=/2.
  reflexivity  @ X<=X <=> true.
  antisymmetry @ X<=Y, Y<=X <=> X=Y.
  idempotence  @ X<=Y \ X<=Y <=> true.
  transitivity @ X<=Y, Y<=Z ==> X<=Z.
  ```

- Example execution:

  - Goal: A<=B, B<=C, C<=A
  - Store: A<=B, B<=C
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X<=X <-> true.
antisymmetry @ X<=Y, Y<=X <-> X=Y.
idempotence @ X<=Y \ X<=Y <-> true.
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```

Example execution:

- Goal: A<=B, B<=C, C<=A
- Store: A<=B, B<=C, A<=C

Example 1: less-or-equal solver
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint less_or_equal/2.
reflexivity @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: A <= B, B <= C, C <= A
- Store: A <= B, B <= C, A <= C, C <= A
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint \</.2.

reflexivity  @ X \<X  \<true.
antisymmetry  @ X \<Y, Y \<X  \<X=Y.
idempotence   @ X \<Y \<X \<Y  \<true.
transitivity  @ X \<Y, Y \<Z  \<X \<Z.
```

Example execution:

- Goal: A \<B, B \<C, C \<A
- Store: A \<B, B \<C, A \<C, C \<A
Example 1: less-or-equal solver

Typical “solver” CHR program:
:- chr_constraint <=/2.
reflexivity @ X≤X <=> true.
antisymmetry @ X≤Y, Y≤X <=> X=Y.
idempotence @ X≤Y \ X≤Y <=> true.
transitivity @ X≤Y, Y≤Z ==> X≤Z.

Example execution:
- Goal: A≤B, B≤C, C≤A
- Store: A≤B, B≤C, A=C
Example 1: less-or-equal solver

Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity  @ X ≤ X <=> true.
antisymmetry @ X ≤ Y, Y ≤ X <=> X = Y.
idempotence  @ X ≤ Y \ X ≤ Y <=> true.
transitivity @ X ≤ Y, Y ≤ Z ==> X ≤ Z.
```

Example execution:

- Goal: A ≤ B, B ≤ C, C ≤ A
- Store: A ≤ B, B ≤ A, A = C
Example 1: less-or-equal solver

- Typical “solver” CHR program:
  ```prolog
  :- chr_constraint <=/2.
  reflexivity  @ X <= X <=/2 true.
  antisymmetry @ X <= Y, Y <= X => X=Y.
  idempotence @ X <= Y \ X <= Y <=/2 true.
  transitivity @ X <= Y, Y <= Z => X <= Z.
  ```

- Example execution:
  - Goal: A <= B, B <= C, C <= A
  - Store: A <= B, B <= A, A = C
Introduction to CHR
Analysis and Optimizing Compilation
Complexity
Extensions of CHR

Example 1: less-or-equal solver

- Typical “solver” CHR program:
  
  ```prolog
  :- chr_constraint <=/2.
  reflexivity @ X <= X <=> true.
  antisymmetry @ X <= Y, Y <= X <=> X = Y.
  idempotence @ X <= Y \ X <= Y <=> true.
  transitivity @ X <= Y, Y <= Z ==> X <= Z.
  ```

- Example execution:
  - Goal: A <= B, B <= C, C <= A
  - Store: A = B, A = C
Typical “solver” CHR program:

```prolog
:- chr_constraint <=/2.
reflexivity @ X <= X <=> true.
antisymmetry @ X <= Y, Y <= X <=> X = Y.
idempotence @ X <= Y \ X <= Y <=> true.
transitivity @ X <= Y, Y <= Z ==> X <= Z.
```

Example execution:

- Goal: A <= B, B <= C, C <= A
- Store: A = B, A = C      ← answer
Example 2: Prime number generation

Typical “general-purpose” CHR program:

```prolog
:- chr_constraint upto/1, prime/1.
upto(1) <=> true.
upto(N) <=> N > 1 | prime(N), upto(N-1).
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

- First two rules implement a loop, generating a sequence 
  `prime(2), ..., prime(n)`
- Third rule filters out the non-prime numbers:
  - if I divides J, then J is not a prime
  - the rule removes such non-primes
Overview

1. Introduction to CHR
   - CHR researchers
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimizing Compilation
   - CHR systems
   - Guard reasoning
   - Memory reuse

3. Complexity
   - Asymptotic complexities
   - Constant factors
   - Other declarative languages

4. Extensions of CHR
   - Negation
   - Aggregates
Recent trends:

- Many new CHR(Prolog) systems (every self-respecting Prolog has CHR now)
- Concurrent CHR(Haskell) systems
- Fast systems in Java & C
K.U.Leuven CHR system
- State-of-the-art CHR system
- for hProlog, SWI-Prolog, XSB, YAP, B-Prolog, ...

Goal: make it even better!
Guard reasoning

- Papers at WCLP’05, ICLP’05, submitted to LNAI special issue
- **Refined operational semantics:** activates constraints depth-first left-to-right; searches for matching rules by trying the occurrences in textual order
- This strategy adds lots of implicit guards, e.g.
  \[ X \leq X \iff \text{true}. \]
  \[ X \leq Y, \ Y \leq X \iff X \neq Y \mid X=Y. \]
  \[ X \leq Y \mid X \leq Y \iff X \neq Y \mid \text{true}. \]
  \[ X \leq Y, \ Y \leq Z \implies X \neq Y, \ Y \neq Z, \ X \neq Z \mid X \leq Z. \]
- Programmers remove implied guards to improve performance
- Problem for logical reading of a rule
- Solution: keep all necessary guards in program, let compiler remove redundant guards
Guard reasoning

- Papers at WCLP’05, ICLP’05, submitted to LNAI special issue
- **Refined operational semantics**: activates constraints
depth-first left-to-right; searches for matching rules by trying
the occurrences in textual order

- This strategy adds lots of implicit guards, e.g.
  - $X \leq X \iff \text{true}$.
  - $X \leq Y, Y \leq X \iff X \neq Y \lor X = Y$.
  - $X \leq Y \lor X \leq Y \iff X \neq Y \lor \text{true}$.
  - $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \lor X \leq Z$.

- Programmers remove implied guards to improve performance

- Problem for logical reading of a rule

- Solution: keep all necessary guards in program, let compiler
  remove redundant guards
Papers at WCLP’05, ICLP’05, submitted to LNAI special issue

**Refined operational semantics:** activates constraints
depth-first left-to-right; searches for matching rules by trying
the occurrences in textual order

This strategy adds lots of implicit guards, e.g.

\[ \begin{align*}
X \leq X & \iff \text{true}. \\
X \leq Y, \ Y \leq X & \iff \ X \neq Y \lor X=Y. \\
X \leq Y \land X \leq Y & \iff \ X \neq Y \lor \text{true}. \\
X \leq Y, \ Y \leq Z & \implies \ X \neq Y, \ Y \neq Z, \ X \neq Z \lor X \leq Z.
\end{align*} \]

Programmers remove implied guards to improve performance

Problem for logical reading of a rule

Solution: keep all necessary guards in program, let compiler
remove redundant guards
Guard reasoning

- Papers at WCLP’05, ICLP’05, submitted to LNAI special issue
- **Refined operational semantics**: activates constraints depth-first left-to-right; searches for matching rules by trying the occurrences in textual order
- This strategy adds lots of implicit guards, e.g.
  \[
  X \leq X \iff \text{true}.
  \]
  \[
  X \leq Y, Y \leq X \iff X \neq Y \lor X = Y.
  \]
  \[
  X \leq Y \setminus X \leq Y \iff X \neq Y \lor \text{true}.
  \]
  \[
  X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \lor X \leq Z.
  \]
- Programmers remove implied guards to improve performance
- Problem for logical reading of a rule
- Solution: keep all necessary guards in program, let compiler remove redundant guards
Papers at WCLP’05, ICLP’05, submitted to LNAI special issue

**Refined operational semantics:** activates constraints depth-first left-to-right; searches for matching rules by trying the occurrences in textual order

This strategy adds lots of implicit guards, e.g.

- \( X \leq X \iff \text{true} \)
- \( X \leq Y, Y \leq X \iff X \neq Y \lor X=Y \)
- \( X \leq Y \lor X \leq Y \iff X \neq Y \lor \text{true} \)
- \( X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \lor X \leq Z \)

Programmers remove implied guards to improve performance

Problem for logical reading of a rule

Solution: keep all *necessary* guards in program, let compiler remove redundant guards
Guard reasoning

- Papers at WCLP’05, ICLP’05, submitted to LNAI special issue
- **Refined operational semantics**: activates constraints depth-first left-to-right; searches for matching rules by trying the occurrences in textual order
- This strategy adds lots of implicit guards, e.g.
  \[
  X \leq X \iff \text{true}.
  \]
  \[
  X \leq Y, \ Y \leq X \iff X \neq Y \lor X=Y.
  \]
  \[
  X \leq Y \setminus X \leq Y \iff X \neq Y \lor \text{true}.
  \]
  \[
  X \leq Y, \ Y \leq Z \implies X \neq Y, \ Y \neq Z, \ X \neq Z \lor X \leq Z.
  \]
- Programmers remove implied guards to improve performance
- Problem for logical reading of a rule
- Solution: keep all *necessary* guards in program, let compiler remove redundant guards
Use type and mode information to improve optimizations

Details

Optional type/mode declarations:

```lean
:- chr_type list(T) ---> [] ; [T | list(T)].
:- chr_constraint sum(+list(int), ?int).
```

Mode: + for ground arguments, ? for unknown mode.

Type: built-in types like any, int, natural, ... new types can be defined using (generic) type definitions

Hash-table store for ground constraints

Mode/type also useful in guard/continuation optimization

E.g. first argument of sum/2 is ground list:

```lean
sum([],S) <=> S=0.
sum([X|Xs],S) <=> sum(Xs,T), S is X+T.
```

⇒ sum/2 is never-stored!
Introduction to CHR

Analysis and Optimizing Compilation

Complexity

Extensions of CHR

Jon Sneyers

Constraint Handling Rules: recent research
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Optimize</th>
<th># clauses</th>
<th># lines</th>
<th>Runtime (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>no</td>
<td>3</td>
<td>10</td>
<td>5.03 (100)</td>
</tr>
<tr>
<td>(10000,5000)</td>
<td>type</td>
<td>2</td>
<td>6</td>
<td>4.49 (89)</td>
</tr>
<tr>
<td>nrev</td>
<td>no</td>
<td>6</td>
<td>20</td>
<td>13.97 (100)</td>
</tr>
<tr>
<td>(30,50000)</td>
<td>type</td>
<td>4</td>
<td>11</td>
<td>8.44 (60)</td>
</tr>
<tr>
<td>dfsearch</td>
<td>no</td>
<td>4</td>
<td>16</td>
<td>37.58 (100)</td>
</tr>
<tr>
<td>(16,500)</td>
<td>yes</td>
<td>4</td>
<td>15</td>
<td>31.63 (84)</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>3</td>
<td>11</td>
<td>29.97 (80)</td>
</tr>
<tr>
<td>bool_chain</td>
<td>no</td>
<td>180</td>
<td>2861</td>
<td>12.8 (100)</td>
</tr>
<tr>
<td>(200)</td>
<td>yes</td>
<td>147</td>
<td>2463</td>
<td>7.0 (55)</td>
</tr>
<tr>
<td>fib</td>
<td>no</td>
<td>10</td>
<td>154</td>
<td>11.2 (100)</td>
</tr>
<tr>
<td>(22)</td>
<td>yes</td>
<td>9</td>
<td>125</td>
<td>8.5 (76)</td>
</tr>
<tr>
<td>leq</td>
<td>no</td>
<td>18</td>
<td>218</td>
<td>14.1 (100)</td>
</tr>
<tr>
<td>(60)</td>
<td>yes</td>
<td>13</td>
<td>162</td>
<td>11.7 (83)</td>
</tr>
</tbody>
</table>
Memory reuse

- Paper at ICLP’06: suspension reuse & in-place updates
  - “Suspension”: internal representation of CHR constraints as a Prolog term: e.g. $A \leq B$ could be represented as a term $S = \text{suspension}(37, \text{stored},’\leq/2_\_0’(A,B,S),\text{nohist},<,A,B)$
  - Constraint store is implemented using hashtables and/or arrays and/or lists and/or attributed variables
  - at constraint insertion: create new suspension term; insert into data structures
  - at constraint removal: delete suspension from data structures; suspension becomes garbage
**Time Results**

### Time:

- **Suspension reuse:**
  - often overhead > gain
  - net result: +12% to −60%

- **In-place updates:**
  - not always applicable
  - net result: +3% to −45%

- **Combining both:**
  - usually slightly worse than only in-place
**Time Results**

- **Suspension reuse:**
  - often overhead > gain
  - net result: +12% to −60%

- **In-place updates:**
  - not always applicable
  - net result: +3% to −45%

- **Combining both:**
  - usually slightly worse than only in-place
Time Results

TIME:

▸ Suspension reuse:
  ▸ often overhead > gain
  ▸ net result: +12% to −60%

▸ In-place updates:
  ▸ not always applicable
  ▸ net result: +3% to −45%

▸ Combining both:
  ▸ usually slightly worse than only in-place

Jon Sneyers  
Constraint Handling Rules: recent research
Space Results

**SPACE:**

- **Suspension reuse:**
  - cost very small
  - net result: 0% to −80%

- **In-place updates:**
  - ram simul: $O(n) \rightarrow O(1)$
  - net result: 0% to −100%

- **Combining both:**
  - best of both
    (or slightly better)
**Space:**

- **Suspension reuse:**
  - cost very small
  - net result: 0% to −80%

- **In-place updates:**
  - \( \text{ram simul}: O(n) \rightarrow O(1) \)
  - net result: 0% to −100%

- **Combining both:**
  - best of both
    (or slightly better)
**Space Results**

### SPACE:

- **Suspension reuse:**
  - Cost very small
  - Net result: 0% to −80%

- **In-place updates:**
  - Ram simul: $O(n) \rightarrow O(1)$
  - Net result: 0% to −100%

- **Combining both:**
  - Best of both (or slightly better)
Overview

1. Introduction to CHR
   - CHR researchers
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimizing Compilation
   - CHR systems
   - Guard reasoning
   - Memory reuse

3. Complexity
   - Asymptotic complexities
   - Constant factors
   - Other declarative languages

4. Extensions of CHR
   - Negation
   - Aggregates

Jon Sneyers
Constraint Handling Rules: recent research
Computational power and complexity of CHR

- Paper at CHR’05 workshop; paper submitted to TOPLAS
- *CHR machine* vs. TM and RAM
Complexity of CHR

- Big-step “CHR machine”
  \( \text{step} = \omega_t \text{ transition} \approx 1 \text{ rule application} \)

- CHR machine can be simulated on RAM machine
  \([\text{duh, this is what CHR compilers are for}]\)

- RAM machine can be simulated on CHR machine
  \([\text{easy: write a RAM simulator in CHR}]\)

- RAM machine can simulate a CHR machine which simulates a RAM machine \(X\)
  \([\text{of course, follows from above}]\)

  \(\ldots\) with the same time and space complexity as \(X\)

  \([\text{this is the tricky part: CHR compiler has to be good for this}]\)

- Conclusion: (in principle,) you can do everything in CHR, with the right time/space complexity
Complexity of CHR

- Big-step “CHR machine”
  (step = \( \omega_t \) transition \( \approx \) 1 rule application)

- CHR machine can be simulated on RAM machine
  \( [\text{duh, this is what CHR compilers are for}] \)

- RAM machine can be simulated on CHR machine
  \( [\text{easy: write a RAM simulator in CHR}] \)

- RAM machine can simulate a CHR machine which simulates a
  RAM machine \( X \)
  \( [\text{of course, follows from above}] \)
  \( \ldots \) with the same time and space complexity as \( X \)
  \( [\text{this is the tricky part: CHR compiler has to be good for this}] \)

- Conclusion: (in principle,) you can do everything in CHR,
  with the right time/space complexity
Big-step “CHR machine”
(step = \( \omega_t \) transition \( \approx 1 \) rule application)

CHR machine can be simulated on RAM machine
[duh, this is what CHR compilers are for]

RAM machine can be simulated on CHR machine
[easy: write a RAM simulator in CHR]

RAM machine can simulate a CHR machine which simulates a RAM machine \( X \)
[of course, follows from above]

... with the same time and space complexity as \( X \)
[this is the tricky part: CHR compiler has to be good for this]

Conclusion: (in principle,) you can do everything in CHR, with the right time/space complexity
Complexity of CHR

- Big-step “CHR machine”
  \( \text{step} = \omega_t \text{ transition} \approx 1 \text{ rule application} \)

- CHR machine can be simulated on RAM machine
  \( \text{[duh, this is what CHR compilers are for]} \)

- RAM machine can be simulated on CHR machine
  \( \text{[easy: write a RAM simulator in CHR]} \)

- RAM machine can simulate a CHR machine which simulates a RAM machine \( X \)
  \( \text{[of course, follows from above]} \)
  \( \ldots \text{with the same time and space complexity as } X \)
  \( \text{[this is the tricky part: CHR compiler has to be good for this]} \)

- Conclusion: (in principle,) you can do everything in CHR, with the right time/space complexity
Big-step “CHR machine”
(step = $\omega_t$ transition $\approx$ 1 rule application)

CHR machine can be simulated on RAM machine
[duh, this is what CHR compilers are for]

RAM machine can be simulated on CHR machine
[easy: write a RAM simulator in CHR]

RAM machine can simulate a CHR machine which simulates a RAM machine $X$
[of course, follows from above]
... with the same time and space complexity as $X$
[this is the tricky part: CHR compiler has to be good for this]

Conclusion: (in principle,) you can do everything in CHR, with the right time/space complexity
\( m/2 \): memory cells (address,value) \( pc/1 \): program counter (label)

\[
\text{prog}(L,\text{init},A), \ m(A,B) \ \text{\textbackslash} \ m(B,0), \ pc(L+1).
\]

\[
\text{prog}(L,\text{cnst},B,A) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,B), \ pc(L+1).
\]

\[
\text{prog}(L,\text{add},B,A), \ m(B,Y) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,X+Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{sub},B,A), \ m(B,Y) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,X-Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{mul},B,A), \ m(B,Y) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,X*Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{div},B,A), \ m(B,Y) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,X/Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{mov},B,A), \ m(B,Y) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{imv},B,A), \ m(B,C), \ m(C,Y) \ \text{\textbackslash} \ m(A,X), \ pc(L) \Rightarrow m(A,Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{mvi},B,A), \ m(B,Y), \ m(A,C) \ \text{\textbackslash} \ m(C,X), \ pc(L) \Rightarrow m(C,Y), \ pc(L+1).
\]

\[
\text{prog}(L,\text{jmp},A) \ \text{\textbackslash} \ pc(L) \Rightarrow pc(A).
\]

\[
\text{prog}(L,\text{cjmp},A,J), \ m(A,0) \ \text{\textbackslash} \ pc(L) \Rightarrow pc(J).
\]

\[
\text{prog}(L,\text{cjmp},A,J), \ m(A,X) \ \text{\textbackslash} \ pc(L) \Rightarrow X \neq 0 \ | \ pc(L+1).
\]

\[
\text{prog}(L,\text{halt}) \ \text{\textbackslash} \ pc(L) \Rightarrow \text{true}.
\]
A \xrightarrow{X} B : a T-time, S-space A can be simulated on a X-time B.
Constant factors

- Paper at WLP’06 (*Dijkstra’s algorithm with Fibonacci heaps: An executable description in CHR*)
- CHR implementation of Fibonacci heaps and Dijkstra’s algorithm
- Compare CHR implementation with C implementation
- C is “only” 10 times faster
Also in paper submitted to TOPLAS

Try to “port” the complexity result to other declarative languages

Write RAM simulator in other languages

Languages we have considered:

- Logic programming: Prolog
- Functional programming: Haskell
- Term-rewrite systems: Maude
- Rule engines: Jess
Overview

1. Introduction to CHR
   - CHR researchers
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimizing Compilation
   - CHR systems
   - Guard reasoning
   - Memory reuse

3. Complexity
   - Asymptotic complexities
   - Constant factors
   - Other declarative languages

4. Extensions of CHR
   - Negation
   - Aggregates
CHR⁻: CHR with negation as absence

- Paper at CHR’06
- Syntax: Head \ Negated_head <= Body
- Operational semantics: rule fires if Head is in the store and Negated_head is not in the store
- Example: person(X) \ married(X) => single(X).
- Can also be done in refined semantics:
  person(X) => check_married(X).
  married(X) \ check_married(X) => true.
  check_married(X) => single(X).
- CHR⁻ allows shorter, more readable programs
- CHR⁻ also triggers rules on removal of negated heads (hard to do manually)
- However, programming in CHR⁻ seems to be tricky
Introduction to CHR
Analysis and Optimizing Compilation
Complexity
Extensions of CHR

Aggregates in CHR

- Paper at CHR’07 and LOPSTR’07
- Add aggregates to CHR: count, sum, max, avg, ...
- Negation is special case of this:
  \[ \text{not}(NH) \equiv \text{count}(NH, 0) \]
- Also user-defined aggregates, nested aggregates
- Example: a graph is Eulerian if \( \forall \) nodes, in-degree = out-degree:
  \[
  \text{forall}(\text{node}(N), (\text{nb}(\text{edge}(N, _), X), \text{nb}(\text{edge}( _, N), X))) \equiv \text{eulerian_graph}.
  \]
- Manually (without aggregates) this takes 8 rules and 3 auxiliary constraints
- Overhead w.r.t manual (hence specialized) versions is acceptable (1.5 - 3)
Other recent CHR research: see new survey (submitted to TPLP)

Future and ongoing work:
- Properties of generalized CHR machines
  - non-deterministic CHR machines
  - self-modifying CHR machines
- Execution strategies; generalized confluence
- Join ordering
- ...

Questions?
Conclusion

- Other recent CHR research: see new survey (submitted to TPLP)
- Future and ongoing work:
  - Properties of generalized CHR machines
    - non-deterministic CHR machines
    - self-modifying CHR machines
  - Execution strategies; generalized confluence
  - Join ordering
  - ...

Questions?
Other recent CHR research: see new survey (submitted to TPLP)

Future and ongoing work:
- Properties of generalized CHR machines
  - non-deterministic CHR machines
  - self-modifying CHR machines
- Execution strategies; generalized confluence
- Join ordering
- ...

Questions?
Now come some extra slides

Read them by clicking on “Details” buttons in the main slides
Use type and mode information to improve optimizations

Optional type/mode declarations:

:- chr_type list(T) ---> [] ; [T | list(T)].
:- chr_constraint sum(+list(int), ?int).

Mode: + for ground arguments, ? for unknown mode.

Type: built-in types like any, int, natural, ...
new types can be defined using (generic) type definitions

Hash-table store for ground constraints

Mode/type also useful in guard/continuation optimization

E.g. first argument of sum/2 is ground list:

sum(L,S) <=> L=[] | S=0.
sum(L,S) <=> L=[X|Xs] | sum(Xs,T), S is X+T.

⇒ sum/2 is never-stored!
Use type and mode information to improve optimizations

Optional type/mode declarations:

\[-\text{chr\_type~list}(T) \rightarrow \text{[] ; [T | list}(T])\].
\[-\text{chr\_constraint~sum}(+\text{list}(\text{int}), ?\text{int})\].

Mode: + for ground arguments, ? for unknown mode.

Type: built-in types like any, int, natural, ... new types can be defined using (generic) type definitions

Hash-table store for ground constraints

Mode/type also useful in guard/continuation optimization

E.g. first argument of \text{sum}/2 is ground list:
\[
\text{sum}(\text{L}, \text{S}) \iff \text{L=[]} \mid \text{S=0.}
\]
\[
\text{sum}(\text{L}, \text{S}) \iff \text{L=}[\text{X}\mid\text{Xs}], \text{sum(Xs,T)}, \text{S is X+T.}
\]

\[\Rightarrow \text{sum}/2 \text{~is~never-stored!}\]
Use type and mode information to improve optimizations

Optional type/mode declarations:

\[
\begin{align*}
&\text{:- chr_type list(T) ---> [] ; [T | list(T)].} \\
&\text{:- chr_constraint sum(+list(int), ?int).}
\end{align*}
\]

Mode: + for ground arguments, ? for unknown mode.

Type: built-in types like any, int, natural, …

new types can be defined using (generic) type definitions

Hash-table store for ground constraints

Mode/type also useful in guard/continuation optimization

E.g. first argument of \(\text{sum}/2\) is ground list:

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

\(\Rightarrow \text{sum/2 is never-stored!}\)
Use type and mode information to improve optimizations

Optional type/mode declarations:

```prolog
:- chr_type list(T) ---> [] ; [T | list(T)].
:- chr_constraint sum(+list(int), ?int).
```

Mode: + for ground arguments, ? for unknown mode.

Type: built-in types like any, int, natural, ... new types can be defined using (generic) type definitions

Hash-table store for ground constraints

Mode/type also useful in guard/continuation optimization

E.g. first argument of sum/2 is ground list:

```prolog
sum(L,S) <=> L=[] | S=0.
sum(L,S) <=> L=[X|Xs], sum(Xs,T), S is X+T.
```

$\Rightarrow$ sum/2 is never-stored! $\Rightarrow$ much cleaner generated code
- Paper at ICLP’06: suspension reuse & in-place updates
- “Suspension”: internal representation of CHR constraints as a Prolog term: e.g. $A \leq B$ could be represented as a term $S = \text{suspension}(37, \text{stored},'\leq/2\_0'(A,B,S),\text{nohist},\leq,A,B)$
- Constraint store is implemented using hashtables and/or arrays and/or lists and/or attributed variables
  - at constraint insertion: create new suspension term; insert into data structures
  - at constraint removal: delete suspension from data structures; suspension becomes garbage
Memory reuse

- Paper at ICLP’06: suspension reuse & in-place updates
- “Suspension”: internal representation of CHR constraints as a Prolog term: e.g. $A \leq B$ could be represented as a term $S =$ suspension(37,stored,’$\leq$/2_0’(A,B,S),nohist,$\leq$,A,B)
- Constraint store is implemented using hashtables and/or arrays and/or lists and/or attributed variables
- at constraint insertion: create new suspension term; insert into data structures
- at constraint removal: delete suspension from data structures; suspension becomes garbage
Paper at ICLP’06: suspension reuse & in-place updates

“Suspension”: internal representation of CHR constraints as a Prolog term: e.g. $A \leq B$ could be represented as a term $S =$ suspension(37, stored, ’\leq/2_0’(A,B,S),nohist,\leq,A,B)

Constraint store is implemented using hashtables and/or arrays and/or lists and/or attributed variables

at constraint insertion: create new suspension term; insert into data structures

at constraint removal: delete suspension from data structures; suspension becomes garbage
Suspensions

- Constraint insertion
- Constraint removal
- New constraint insertion

Constraints:
- arg1
- arg34
- arg5

Constraint store for c/5

Jon Sneyers
Constraint Handling Rules: recent research
Suspensions

- Constraint insertion
- Constraint removal
- New constraint insertion
Suspensions

- Constraint insertion
- Constraint removal
- New constraint insertion
Typical pattern:

\[ \text{update}(\text{Key}, \text{NewV}), \quad \text{item}(\text{Key}, \text{OldV}) \iff \text{item}(\text{Key}, \text{NewV}). \]

E.g.

\[ \text{upd25}(A, X, Y), \quad \text{c}(A, B, C, D, E) \iff \text{c}(A, X, C, D, Y). \]
**In-place updates**

- Typical pattern:
  \[
  \text{update}(\text{Key}, \text{NewV}), \quad \text{item}(\text{Key}, \text{OldV}) \iff \text{item}(\text{Key}, \text{NewV}) .
  \]

- e.g. \(\text{upd25}(A, X, Y), \quad \text{c}(A, B, C, D, E) \iff \text{c}(A, X, C, D, Y) .\)

---

Jon Sneyers  
Constraint Handling Rules: recent research
Pattern of in-place updates is often used... but: many other ways to remove/insert a constraint

For example:

\[
\begin{align*}
\text{candidate}(1) & \iff \text{true}. \\
\text{candidate}(N) & \iff N>1 \lor \text{prime}(N), \text{candidate}(N-1). \\
\text{prime}(I) \not\iff \text{prime}(J) & \iff J \mod I = 0 \lor \text{true}.
\end{align*}
\]

Suspension reuse = dynamic in-place updates

Reuse of an old idea:

- maintain a cache of old suspensions
- when constraint is removed, put suspension in cache and keep constraint in data structures (but mark it)
- when a suspension is created, first check cache
- dynamically check which data structures to fix
Pattern of in-place updates is often used…
…but: many other ways to remove/insert a constraint

For example:

```
candidate(1) <=> true.
candidate(N) <=> N>1 | prime(N), candidate(N-1).
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```


Suspension reuse = dynamic in-place updates

Reuse of an old idea:

- maintain a cache of old suspensions
- when constraint is removed, put suspension in cache and keep constraint in data structures (but mark it)
- when a suspension is created, first check cache
- dynamically check which data structures to fix
Suspension reuse

- Constraint insertion
- Constraint removal
- New constraint insertion
Suspension reuse

- Constraint insertion
- Constraint removal
- New constraint insertion
Suspension reuse

Constraint insertion

Constraint removal

New constraint insertion
Benchmark: Dijkstra’s shortest path algorithm

- Runtime in seconds (logscale)
- Input size / 1024 (logscale)

- $O(n)$
- $O(n^2)$

- C
- CHR [2]
- CHR [1]
- CHR [0]
- SICStus
- Java

Jon Sneyers  
Constraint Handling Rules: recent research
Benchmark: RAM simulator [nested loop]

Runtime divided by $O(n^2)$ (logscale)

- CHR [2]
- CHR [1]
- CHR [0]
- Prolog
- Haskell [strict]
- Maude [Map]
- Jess

Jon Sneyers  
Constraint Handling Rules: recent research