Constraint Handling Rules: Analysis, Optimization, Complexity, Extensions

Jon Sneyers
K.U. Leuven, Belgium

DTAI seminar
May 8, 2007
Overview

1. Introduction to CHR
   - The CHR team
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimization
   - Guard reasoning
   - Memory reuse

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

4. Extensions of CHR
   - Negation
   - Aggregates

CHR: Analysis, Optimization, Complexity, Extensions
1 Introduction to CHR
   - The CHR team
   - Syntax, semantics, results
   - Examples

2 Analysis and Optimization
   - Guard reasoning
   - Memory reuse

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

4 Extensions of CHR
   - Negation
   - Aggregates
Introduction to CHR
Complexity
Extensions of CHR

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]
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]
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? (from summer 2007?)
- Master thesis on termination
- Pieter Wuille? (from summer 2007?)
- Master thesis on 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)

CHR(Java)

Paolo Pilozzi
(since summer 2006)

termination

Dean Voets?
(from summer 2007?)

Leslie De Koninck

INCLP(R)

Pieter Wuille?
(from summer 2007?)

Master thesis on 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)
  - CHR(Java)
- Paolo Pilozzi (since summer 2006)
  - termination
- Dean Voets? (from summer 2007?)
  - Master thesis on termination
- Pieter Wuille? (from summer 2007?)
  - Master thesis on 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)
  - INCLP(R)
  - CHR(Java)
  - Paol Pilozzi (since summer 2006)
  - termination
  - Dean Voets? (from summer 2007?)
    - Master thesis on termination
  - Pieter Wuille? (from summer 2007?)
    - Master thesis on 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? (from summer 2007?)
  Master thesis on termination
- Pieter Wuille? (from summer 2007?)
  Master thesis on 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? (from summer 2007?)

Master thesis on termination

Pieter Wuille? (from summer 2007?)

Master thesis on CHR(C)
**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:** \(h \iff g | b\)
- **Propagation:** \(h \implies g | b\)
- **Simpagation:** \(h_1 \setminus h_2 \iff g | b\)

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:** \(g \rightarrow (h \leftarrow b)\)
- **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 \(h_2\)
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: $g \rightarrow (h \leftarrow b)$
- 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 $h_2$
**Syntax and semantics of CHR** on 1 slide

- **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: \(h \implies g | b\) 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: \(g \rightarrow (h \leftrightarrow b)\)
  - 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**: \( h \leftrightarrow g|b \)
- **Propagation**: \( h \rightarrow g|b \)
- **Simpagation**: \( h_1\backslash h_2 \leftrightarrow g|b \)

**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 \( h_2 \)

**Examples**

```plaintext
I CHR(X) where X is host-language e.g. X = Prolog
I CHR constraints, defined in CHR program e.g. ≤
I Built-in (host-language) constraints, theory CT e.g. =
I Syntax: CHR program consists of rules
I Simplification: h ↔ g|b e.g. A≤B, B≤A ↔ A=B.
I Propagation: h → g|b e.g. A≤B, B≤C → A≤C.
I Simpagation: h₁\backslash h₂ ↔ g|b e.g. A≤B \ A≤B ↔ true.
I head h, guard g and body b are conjunctions of constraints
  (h: only CHR constraints; g: only host-language constraints)
I Logical semantics: rules define theory \( \mathcal{P} \)
I Simplification: g → (h ↔ b)
I Propagation: g → (h → b)
I Simpagation: g → (h₁ → (h₂ ↔ b))
I Operational semantics: rules manipulate constraint store
I if h is in constraint store and g holds, then add b
I Simplification: remove h;  
  Propagation: keep h;  
  Simpagation: keep h₁, remove h₂
```
**Syntax and semantics of CHR on 1 slide**

- **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: $h \iff g|b$ 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: $g \rightarrow (h \leftrightarrow b)$
  - 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 $h_2$
Some properties and results

- 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 $P, CT \models G \leftarrow A$

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

- **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:
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
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
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
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
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:

```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:
  
  ```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

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: \(A \leq B, B \leq C, C \leq A\)
- Store: \(A \leq B, B \leq A, A = C\)
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 <=> 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: \(A \leq B, B \leq C, C \leq A\)
- Store: \(A = B, A = C\) ← answer
Example 2: Dijkstra’s shortest path algorithm

Typical “general-purpose” CHR program:
:- chr_constrain edge/3, source/1, dist/2, scan/1,
    label/2, newlabel/2.

source(A) <=> label(A,0), scan(A).
scan(A) \ label(A,D) <=> dist(A,D).
scan(A), dist(A,D), edge(A,B,W) ==> newlabel(B,D+W).

dist(B,_) \ newlabel(B,L) <=> true.
label(B,X) \ newlabel(B,L) <=> L >= X | true.
label(B,X), newlabel(B,L) <=> label(B,L), decr_key(B,L).
newlabel(B,L) <=> label(B,L), insert(B,L).
scan(A) <=> extract_min(B,_) | scan(B).
scan(_) <=> true.
Introduction to CHR
- The CHR team
- Syntax, semantics, results
- Examples

Analysis and Optimization
- Guard reasoning
- Memory reuse

Complexity
- Asymptotic complexities
- Constant factors
- Other declarative languages

Extensions of CHR
- Negation
- Aggregates
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 (*Guard simplification*) and ICLP’05 (*Guard and continuation optimization for occurrence representations*)

- **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.
  
  \[
  X \leq X \iff \text{true}.
  \]
  
  \[
  X \leq Y, \ Y \leq X \iff X \neq Y \ | \ X = Y.
  \]
  
  \[
  X \leq Y \ \& \ X \leq Y \iff X \neq Y \ | \ \text{true}.
  \]
  
  \[
  X \leq Y, \ Y \leq Z \Rightarrow X \neq Y, \ Y \neq Z, \ X \neq Z \ | \ X \leq Z.
  \]

- 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 (*Guard simplification*) and ICLP’05 (*Guard and continuation optimization for occurrence representations*)

**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
Papers at WCLP’05 (*Guard simplification*) and ICLP’05 (*Guard and continuation optimization for occurrence representations*)

**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.

- \( X \leq X \leftrightarrow \text{true} \).
- \( X \leq Y, Y \leq X \leftrightarrow X \neq Y \mid X = Y \).
- \( X \leq Y \setminus X \leq Y \leftrightarrow X \neq Y \mid \text{true} \).
- \( X \leq Y, Y \leq Z \Rightarrow X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z \).

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 (*Guard simplification*) and ICLP’05 (*Guard and continuation optimization for occurrence representations*)

- **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.

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

- 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 (*Guard simplification*) and ICLP’05 (*Guard and continuation optimization for occurrence representations*)

- **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.
  
  \[
  \begin{align*}
  X \leq X & \leftrightarrow \text{true}. \\
  X \leq Y, Y \leq X & \leftrightarrow X \neq Y \lor X=Y. \\
  X \leq Y \lor X \leq Y & \leftrightarrow X \neq Y \lor \text{true}. \\
  X \leq Y, Y \leq Z & \rightarrow X \neq Y, Y \neq Z, X \neq Z \lor X \leq Z.
  \end{align*}
  \]

- 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 (*Guard simplification*) and ICLP’05 (*Guard and continuation optimization for occurrence representations*)
- **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.
  \[
  X \leq X \leftrightarrow \text{true}.
  \]
  \[
  X \leq Y, Y \leq X \leftrightarrow X \neq Y \mid X = Y.
  \]
  \[
  X \leq Y \setminus X \leq X \leftrightarrow X \neq Y \mid \text{true}.
  \]
  \[
  X \leq Y, Y \leq Z \Rightarrow X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.
  \]
- 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
Type and mode information

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

\(\Rightarrow\) sum/2 is never-stored!
Type and mode information

- 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.
  ```

⇒ `sum/2` is never-stored!
Type and mode information

- 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.
  ```

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

Optional type/mode declarations:

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

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 sum/2 is ground list:

\[\text{sum}(L,S) \leftrightarrow L=[] \mid S=0.\]
\[\text{sum}(L,S) \leftrightarrow L=[X|Xs], \text{sum}(Xs,T), S \text{ is } X+T.\]

⇒ sum/2 is never-stored!
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:

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

⇒ sum/2 is never-stored! ⇒ much cleaner generated code
Introduction to CHR
Analysis and Optimization
Complexity
Extensions of CHR

Guard reasoning
Memory reuse

Effect on generated code

---

with guard optimization

```prolog
:-use_module(library(chr_runtime)).
:-use_module(library(chr_hashtable_store)).
'attach_sum/2'([],_).
'attach_sum/2'(E|D,C):-get_attr(E,use,B)->A=[C|B],put_attr(E,use,A);put_attr(E,use,[C]),'attach_sum/2'(D,C).
'detach_sum/2'([],_). 'detach_sum/2'(E|D,C):-get_attr(E,use,B)->chr_runtime:sbag_del_element(B,C,A),A=[]>del_attr(E,use);put_attr(E,use,A),true),'detach_sum/2'(D,C).
'$indexed_variables'(C,B):-C=sum(A,_),term_variables(A,B),attach_increment([],_),attach_increment([E|D],C):-chr_runtime:merge_attributes(D,B,A),put_attr(F,user,A),put_attr(F,user,D),attach_increment(E,D).
attr_unify_hook(G,F):-sort(G,E),(var(F)->(get_attr(F,use,D)->true;D=[]),sort(D,C),chr_runtime:merge_attributes(E,C,B),put_attr(F,use,B),true),chr_runtime:run_suspensions(B);(compound(F)->term_variables(F,A),attach_increment(A,E);true),chr_runtime:run_suspensions(G).
```

without guard optimization

```prolog
:-use_module(library(chr_runtime)).
:-use_module(library(chr_hashtable_store)).
'$indexed_variables'(C,B):-C=sum(A,_),term_variables(A,B),attach_increment([],_),attach_increment([E|D],C):-chr_runtime:merge_attributes(D,B,A),put_attr(F,use,A),put_attr(F,use,D),attach_increment(E,D).
attr_unify_hook(G,F):-sort(G,E),(var(F)->(get_attr(F,use,D)->true;D=[]),sort(D,C),chr_runtime:merge_attributes(E,C,B),put_attr(F,use,B),true),chr_runtime:run_suspensions(B);(compound(F)->term_variables(F,A),attach_increment(A,E);true),chr_runtime:run_suspensions(G).
```

---

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

Jon Sneyers  
CHR: Analysis, Optimization, Complexity, Extensions
Paper at ICLP’06

“Suspension”: internal representation of CHR constraints as a Prolog term: e.g. \( A \leq B \) could be represented as a term \( S = suspension(37, \text{stored}, '\leq/2__0'(A,B,S),\text{nohist},\leq,A,B) \)

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
Suspensions

- Constraint insertion
- Constraint removal
- New constraint insertion
Suspensions

- Constraint insertion
- Constraint removal
- New constraint insertion
Suspensions

- Constraint insertion
- Constraint removal
- New constraint insertion
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. \(\text{upd25}(A, X, Y),\)
\[
\text{c}(A, B, C, D, E) \iff \text{c}(A, X, C, D, Y).
\]
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
Pattern of in-place updates is often used...
...but: many other ways to remove/insert a constraint

For example:

\[
\begin{align*}
candidate(1) & \iff \text{true}. \\
candidate(N) & \iff N > 1 \lor \text{prime}(N), \text{candidate}(N-1). \\
\text{prime}(I) \setminus \text{prime}(J) & \iff J \mod I = 0 \lor \text{true}.
\end{align*}
\]

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
Suspension reuse

- Pattern of in-place updates is often used...
  ...but: many other ways to remove/insert a constraint
- For example:

  \[
  \begin{align*}
  \text{candidate}(1) & \iff \text{true}. \\
  \text{candidate}(N) & \iff N>1 \mid \text{prime}(N), \text{candidate}(N-1). \\
  \text{prime}(I) \setminus \text{prime}(J) & \iff J \mod I =:= 0 \mid \text{true}.
  \end{align*}
  \]

- 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
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
Introduction to CHR
Analysis and Optimization
Complexity
Extensions of CHR

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**

**TIME:**

- **Suspension reuse:**
  - often overhead $> \text{gain}$
  - 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
### 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 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 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
   - The CHR team
   - Syntax, semantics, results
   - Examples
2. Analysis and Optimization
   - Guard reasoning
   - Memory reuse
3. Complexity
   - Asymptotic complexities
   - Constant factors
   - Other declarative languages
4. Extensions of CHR
   - Negation
   - Aggregates
Paper at CHR’05 workshop; paper submitted to TOPLAS

CHR machine vs. TM and RAM

CHR machine

Turing machine

Random Access Memory machine
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
Big-step “CHR machine”
(step = ω transition ≈ 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
Big-step “CHR machine”
(step = \omega_t \text{ transition} \approx 1 \text{ 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
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
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
\( \text{m}/2 \): memory cells (address, value) \( \text{pc}/1 \): program counter (label)

\text{prog}/\{2,3,4\} : program instructions (label, instruction, arguments)

\begin{align*}
\text{prog}(L, \text{init}, A), & \quad \text{m}(A, B) \\text{\textbackslash pc}(L) \iff \text{m}(B, 0), \text{pc}(L+1). \\
\text{prog}(L, \text{cnst}, B, A) & \quad \text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, B), \text{pc}(L+1).
\end{align*}

\begin{align*}
\text{prog}(L, \text{add}, B, A), & \quad \text{m}(B, Y) \\text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, X+Y), \text{pc}(L+1). \\
\text{prog}(L, \text{sub}, B, A), & \quad \text{m}(B, Y) \\text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, X-Y), \text{pc}(L+1). \\
\text{prog}(L, \text{mul}, B, A), & \quad \text{m}(B, Y) \\text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, X*Y), \text{pc}(L+1). \\
\text{prog}(L, \text{div}, B, A), & \quad \text{m}(B, Y) \\text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, X/Y), \text{pc}(L+1). \\
\text{prog}(L, \text{mov}, B, A), & \quad \text{m}(B, Y) \\text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, Y), \text{pc}(L+1). \\
\text{prog}(L, \text{imv}, B, A), & \quad \text{m}(B, C), \text{m}(C, Y) \\text{\textbackslash m}(A, X), \text{pc}(L) \iff \text{m}(A, Y), \text{pc}(L+1). \\
\text{prog}(L, \text{mvi}, B, A), & \quad \text{m}(B, Y), \text{m}(A, C) \\text{\textbackslash m}(C, X), \text{pc}(L) \iff \text{m}(C, Y), \text{pc}(L+1). \\
\text{prog}(L, \text{jmp}, A) & \quad \text{\textbackslash pc}(L) \iff \text{pc}(A). \\
\text{prog}(L, \text{cjmp}, A, J), & \quad \text{m}(A, 0) \\text{\textbackslash pc}(L) \iff \text{pc}(J). \\
\text{prog}(L, \text{cjmp}, A, J), & \quad \text{m}(A, X) \\text{\textbackslash pc}(L) \iff X \neq 0 \lor \text{pc}(L+1). \\
\text{prog}(L, \text{halt}) & \quad \text{\textbackslash pc}(L) \iff \text{true}. 
\end{align*}
A \xrightarrow{X} B : a T-time, S-space A can be simulated on a X-time B.
- 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
Benchmark: Dijkstra’s shortest path algorithm

- $O(n)$
- $O(n^2)$
- C
- CHR [2]
- CHR [1]
- CHR [0]
- SICStus
- Java

Runtime in seconds (logscale)

Input size / 1024 (logscale)
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
Benchmark: RAM simulator [nested loop]

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

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

- $< 1s$
- $< 10s$
- $< 1m$
- $< 10m$
- $< 1h$

Jon Sneyers  CHR: Analysis, Optimization, Complexity, Extensions
Overview

1. Introduction to CHR
   - The CHR team
   - Syntax, semantics, results
   - Examples

2. Analysis and Optimization
   - Guard reasoning
   - Memory reuse

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

4. Extensions of CHR
   - Negation
   - Aggregates
CHR\textsuperscript{♯}: CHR with negation as absence

- Paper at CHR’06
- Syntax: Head \neg\neg Negated\_head \iff Body
- Operational semantics: rule fires if Head is in the store and Negated\_head is not in the store
- Example: 
  \begin{verbatim}
  person(X) \neg \neg married(X,_) \rightarrow single(X).
  \end{verbatim}
- Without negated heads also possible in refined semantics:
  \begin{verbatim}
  person(X) \rightarrow check\_married(X).
  married(X,_) \neg check\_married(X) \iff true.
  check\_married(X) \iff single(X).
  \end{verbatim}
- CHR\textsuperscript{♯} allows shorter, more readable programs
- CHR\textsuperscript{♯} also triggers rules on removal of negated heads (harder to do manually)
- However, programming in CHR\textsuperscript{♯} seems to be tricky
Ongoing work, maybe CHR’07

Add aggregates to CHR: count, sum, max, avg, ...

Negation is special case of this:
not(NH) ≜ count(NH, 0)

Also user-defined aggregates, nested aggregates

Example: a graph is Eulerian if ∀ nodes, in-degree = out-degree:
forall(node(N), (nb(edge(N, _), X), nb(edge(_, N), X)))

===> 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)
Dijkstra’s algorithm in CHR:

```prolog
:- chr_constraint edge/3, source/1, dist/2, scan/1, label/2, newlabel/2.

source(A) <-> label(A,0), scan(A).
scan(A) \ label(A,D) <-> dist(A,D).
scan(A), dist(A,D), edge(A,B,W) ==> newlabel(B,D+W).

dist(B,_) \ newlabel(B,L) <-> true.
label(B,X) \ newlabel(B,L) <-> L >= X | true.
label(B,X), newlabel(B,L) <-> label(B,L), decr_key(B,L).
newlabel(B,L) <-> label(B,L), insert(B,L).
scan(A) <-> extract_min(B,_) | scan(B).
scan(_) <-> true.

+ some implementation of a priority queue with the operations insert/2, decr_key/2, and extract_min/2.
```
Dijkstra's algorithm in CHR with aggregates:

\[-\text{chr\_constraint} \text{ edge/3, source/1, dist/2, scan/1, label/2, newlabel/2.}\]

\begin{align*}
\text{source}(A) & \iff \text{label}(A,0), \text{scan}(A). \\
\text{scan}(A) & \not\iff \text{label}(A,D) \iff \text{dist}(A,D). \\
\text{scan}(A), \text{dist}(A,D), \text{edge}(A,B,W), \text{no}(\text{dist}(B,_)) & \implies \text{newlabel}(B,D+W). \\
\text{label}(B,X) & \not\iff \text{newlabel}(B,L) \iff L \geq X \mid \text{true}. \\
\text{label}(B,X), \text{newlabel}(B,L) & \iff \text{label}(B,L), \text{decr\_key}(B,L). \\
\text{newlabel}(B,L) & \iff \text{label}(B,L), \text{insert}(B,L). \\
\text{scan}(A) & \iff \text{extract\_min}(B,_) \mid \text{scan}(B). \\
\text{scan}(_) & \iff \text{true}. \\
\end{align*}

+ some implementation of a priority queue with the operations insert/2, decr\_key/2, and extract\_min/2.
Dijkstra’s algorithm in CHR with aggregates:

:- chr_constraint edge/3, source/1, dist/2, scan/1, label/2.

source(A) <=> label(A,0), scan(A).
scan(A) \ label(A,D) <=> dist(A,D).
scan(A), dist(A,D), edge(A,B,W), no(dist(B,_)) ==> label(B,D+W).

label(A,X) \ label(B,Y) <=> X <= Y | true.
scan(A), argmin(L,label(B,L)) | scan(B).
scan(_) <=> true.
- Related work: see papers
- Future work: many interesting possibilities! (see papers)

Questions?