

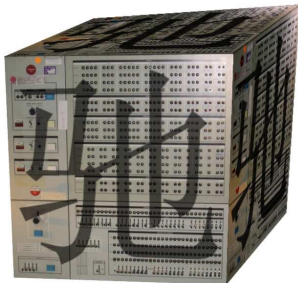
# CONSTRAINT HANDLING RULES: Analysis, Optimization, Complexity, Extensions

Jon Sneyers  
K.U.Leuven, Belgium

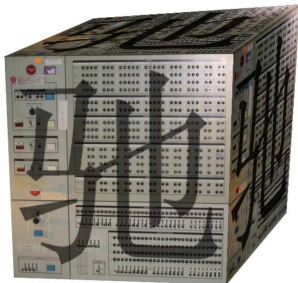


DTAI seminar  
May 8, 2007

驰



- 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



- 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

# Constraint Handling Rules [Frühwirth 1991]

4/37

- ▶ 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]

4/37

- ▶ 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

5/37



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 Pillozzi  
(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

5/37



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

5/37



▶ 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

5/37



▶ 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

5/37



▶ 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

5/37



▶ 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)*

# Syntax and semantics of CHR on 1 slide

6/37

- ▶ **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:  $g \rightarrow (h \leftrightarrow b)$
  - ▶ Propagation:  $g \rightarrow (h \rightarrow b)$
  - ▶ Simpagation:  $g \rightarrow (h_1 \rightarrow (h_2 \leftrightarrow b))$
- ▶ **Operational semantics:** rules manipulate *constraint store*
  - ▶ if  $b$  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

6/37

- ▶ **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:  $g \rightarrow (h \leftrightarrow b)$
  - ▶ 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  $h_2$

# Syntax and semantics of CHR on 1 slide

6/37

- ▶ **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  $h_2$

# Syntax and semantics of CHR on 1 slide

6/37

- ▶ **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  $h_2$

# Syntax and semantics of CHR on 1 slide

6/37

- ▶ **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**

7/37

- ▶ 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  $\mathcal{P}, \mathcal{CT} \models G \leftrightarrow A$
- ▶ **Completeness**. If goal  $G$  terminates and  $\mathcal{P}, \mathcal{CT} \models G \leftrightarrow A$ , then  $G$  has an answer  $A'$  such that  $\mathcal{P}, \mathcal{CT} \models A \leftrightarrow A'$ ,
- ▶ *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  $\mathcal{P} \cup \mathcal{CT}$  is *consistent*.

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A \leq B$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A \leq B, B \leq C$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A \leq B, B \leq C$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A \leq B, B \leq C, A \leq C$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A \leq B, B \leq C, A \leq C, C \leq A$

## Example 1: less-or-equal solver

8/37

- ▶ 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:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A \leq B, B \leq C, A \leq C, C \leq A$

## Example 1: less-or-equal solver

8/37

- ▶ 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:  $A \leq B$ ,  $B \leq C$ ,  $C \leq A$
  - ▶ Store:  $A \leq B$ ,  $B \leq C$ ,  $A=C$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ 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

8/37

- ▶ 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:  $A \leq B$ ,  $B \leq C$ ,  $C \leq A$
  - ▶ Store:  $A \leq B$ ,  $B \leq A$ ,  $A=C$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  
:- chr\_constraint  $\leq$ /2.  
reflexivity @  $X \leq X \iff \text{true}$ .  
antisymmetry @  $X \leq Y, Y \leq X \iff X=Y$ .  
idempotence @  $X \leq Y \setminus X \leq Y \iff \text{true}$ .  
transitivity @  $X \leq Y, Y \leq Z \implies X \leq Z$ .
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A=B, A=C$

## Example 1: less-or-equal solver

8/37

- ▶ Typical “solver” CHR program:  

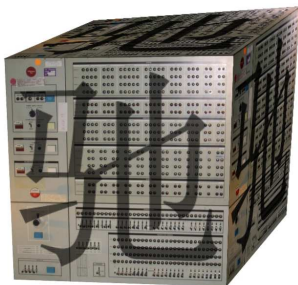
```
:- chr_constraint <= /2.  
reflexivity @ X<=X <=> true.  
antisymmetry @ X<=Y, Y<=X <=> X=Y.  
idempotence @ X<=Y \ X<=Y <=> true.  
transitivity @ X<=Y, Y<=Z ==> X<=Z.
```
- ▶ Example execution:
  - ▶ Goal:  $A \leq B, B \leq C, C \leq A$
  - ▶ Store:  $A=B, A=C$  ← answer

## Example 2: Dijkstra's shortest path algorithm

9/37

- ▶ Typical “general-purpose” CHR program:

```
:- 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.
```



- 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

- ▶ K.U.Leuven CHR system
  - ▶ State-of-the-art CHR system
  - ▶ for hProlog, SWI-Prolog, XSB, YAP, B-Prolog, ...
  - ▶ <http://www.cs.kuleuven.be/~toms/CHR>
- ▶ Goal: make it even better!

## Guard reasoning

12/37

- ▶ 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.  
 $X \leq X \iff \text{true}.$   
 $X \leq Y, Y \leq X \iff X \neq Y \mid X=Y.$   
 $X \leq Y \setminus X \leq Y \iff X \neq Y \mid \text{true}.$   
 $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.$
- ▶ 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

12/37

- ▶ 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.  
 $X \leq X \iff \text{true}.$   
 $X \leq Y, Y \leq X \iff X \neq Y \mid X=Y.$   
 $X \leq Y \setminus X \leq Y \iff X \neq Y \mid \text{true}.$   
 $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.$
- ▶ 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

12/37

- ▶ 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.  
 $X \leq X \iff \text{true}.$   
 $X \leq Y, Y \leq X \iff X \neq Y \mid X=Y.$   
 $X \leq Y \setminus X \leq Y \iff X \neq Y \mid \text{true}.$   
 $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.$
- ▶ 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

12/37

- ▶ 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.  
 $X \leq X \iff \text{true}.$   
 $X \leq Y, Y \leq X \iff X \neq Y \mid X=Y.$   
 $X \leq Y \setminus X \leq Y \iff X \neq Y \mid \text{true}.$   
 $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.$
- ▶ 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

12/37

- ▶ 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.  
 $X \leq X \iff \text{true}.$   
 $X \leq Y, Y \leq X \iff X \neq Y \mid X=Y.$   
 $X \leq Y \setminus X \leq Y \iff X \neq Y \mid \text{true}.$   
 $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.$
- ▶ 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

12/37

- ▶ 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.  
 $X \leq X \iff \text{true}.$   
 $X \leq Y, Y \leq X \iff X \neq Y \mid X=Y.$   
 $X \leq Y \setminus X \leq Y \iff X \neq Y \mid \text{true}.$   
 $X \leq Y, Y \leq Z \implies X \neq Y, Y \neq Z, X \neq Z \mid X \leq Z.$
- ▶ 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

## Type and mode information

13/37

- ▶ 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([],S) <=> S=0.  
sum([X|Xs],S) <=> sum(Xs,T), S is X+T.
```
- ▶  $\Rightarrow$  `sum/2` is never-stored!

## Type and mode information

13/37

- ▶ 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.
```
- ▶  $\Rightarrow$  `sum/2` is never-stored!

## Type and mode information

13/37

- ▶ 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.
```
- ▶  $\Rightarrow$  `sum/2` is never-stored!

## Type and mode information

13/37

- ▶ 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.
```
- ▶  $\Rightarrow$  `sum/2` is never-stored!

## Type and mode information

13/37

- ▶ 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.
```
- ▶  $\Rightarrow$  `sum/2` is never-stored!  $\Rightarrow$  much cleaner generated code

# Effect on generated code

15/37

```
:-use_module(library(chr_runtime)). :-use_module(library(chr_hashtable_store)). 'attach_sum/2' ([],_).
'attach_sum/2' ([E|D],C):- (get_attr(E,user,B)->A=[C|B],put_attr(E,user,A);put_attr(E,user,[C])), 'attach_sum/2' (D,C).
'detach_sum/2' ([],_). 'detach_sum/2' ([E|D],C):- (get_attr(E,user,B)->chr_runtime:sbag_del_element(B,C,A),(A=[]->
del_attr(E,user);put_attr(E,user,A));true), 'detach_sum/2' (D,C). '$indexed_variables'(C,B):-C=sum(A,_),term_variables(
A,B). attach_increment([],_). attach_increment([F|E],D):-chr_runtime:not_locked(F),(get_attr(F,user,C)->sort(C,B),
chr_runtime:merge_attributes(D,B,A),put_attr(F,user,A);put_attr(F,user,D)),attach_increment(E,D).
attr_unify_hook(G,F):-sort(G,E),(var(F)->(get_attr(F,user,D)->true;D=[]),sort(D,C),chr_runtime:merge_attributes(
E,C,B),put_attr(F,user,B),chr_runtime:run_suspensions(B);(compound(F)->term_variables(F,A),attach_increment(A,E);
true),chr_runtime:run_suspensions(G)). activate_constraint(H,G,F,E):-arg(2,F,D),D=mutable(C),chr_runtime:
update_mutable(active,D),(nonvar(E)->true;arg(4,F,B),B=mutable(A),E is A+1,chr_runtime:update_mutable(E,B)),
(compound(C)->term_variables(C,G),chr_runtime:none_locked(G),H=yes;C==removed->chr_indexed_variables(F,G),H=yes;
G=[],H=no). remove_constraint_internal(E,D,C):-arg(2,E,B),B=mutable(A),chr_runtime:update_mutable(removed,B),
(compound(A)->D=[],C=no;A==removed->D=[],C=no;C=yes,chr_indexed_variables(E,D)). insert_constraint_internal(yes,J,I,
H,G,F):-I..[suspension,E,D,H,C,B,G|F],chr_indexed_variables(I,J),chr_runtime:none_locked(J),chr_runtime:
create_mutable(active,D),chr_runtime:create_mutable(O,C),chr_runtime:create_mutable(A,B),chr_runtime:empty_history
(A),chr_runtime:gen_id(E). chr_indexed_variables(C,B):-C=..[_,_,-,-,-,-,_,A|_], '$indexed_variables'(A,B).
'$insert_in_store_sum/2'(D):-chr_runtime:global_term_ref_1(C),(get_attr(C,user,B)->A=[D|B],put_attr(C,user,A);
put_attr(C,user,[D])). '$delete_from_store_sum/2'(D):-chr_runtime:global_term_ref_1(C),(get_attr(C,user,B)->
chr_runtime:sbag_del_element(B,D,A),(A=[]->del_attr(C,user);put_attr(C,user,A));true). '$enumerate_suspensions'(C)
:-chr_runtime:global_term_ref_1(B),get_attr(B,user,A),chr_runtime:sbag_member(C,A). sum(B,A):-'sum/2_0'(B,A,_).
'sum/2_0'(E,D,C):-E=[],!,(var(C)->true;remove_constraint_internal(C,B,A),(A=yes->'$delete_from_store_sum/2'(
C,B,C);true)),D=0. 'sum/2_0'(H,G,F):-nonvar(H),H=[E|D],!(var(F)->true;remove_constraint_internal(
F,C,B),(B=yes->'$delete_from_store_sum/2'(F),'detach_sum/2'(C,F);true)),sum(D,A),G is E+A. 'sum/2_0'(E,D,C):-
(var(C)->insert_constraint_internal(B,A,C,user:'sum/2_0'(E,D,C),sum(E,D),[E,D]));activate_constraint(B,A,C,_),
(B=yes->'$insert_in_store_sum/2'(C),'attach_sum/2'(A,C);true).
```

↓ with guard optimization

without ↑

```
:- use_module(library(chr_runtime)).
:- use_module(library(chr_hashtable_store)).
'$enumerate_suspensions'(_) :- fail.
sum([],A) :- !, A=0.
sum([D|C],B) :- sum(C,A), B is D+A.
```

## Effect on runtime

16/37

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

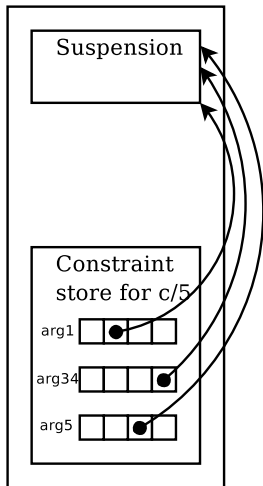
## Memory reuse

17/37

- ▶ 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  $S = \text{suspension}(37, \text{stored}, \text{'}\leq/2\_0\text{'}(A, B, S), \text{nohist}, \leq, A, B)$
- ▶ 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

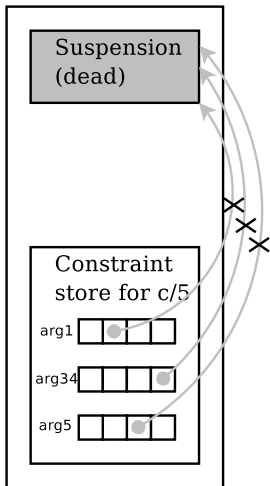
18/37



- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

# Suspensions

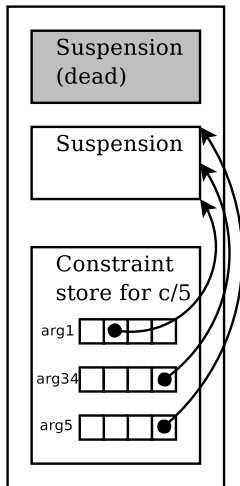
18/37



- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

# Suspensions

18/37



- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

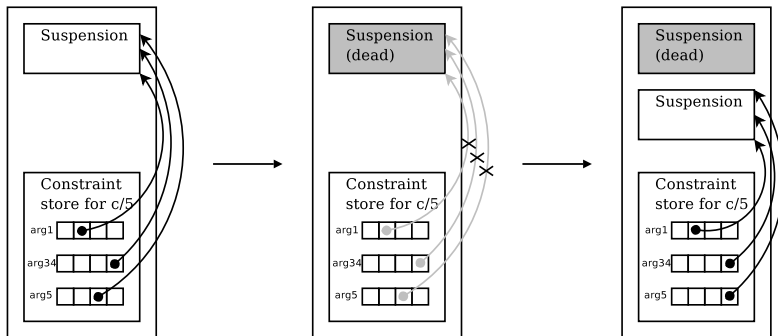
# In-place updates

19/37

- ▶ Typical pattern:

update(Key, NewV), item(Key, OldV)  $\Leftrightarrow$  item(Key, NewV) .

- ▶ e.g. upd25(A, X, Y), c(A, B, C, D, E)  $\Leftrightarrow$  c(A, X, C, D, Y) .



# In-place updates

19/37

- ▶ Typical pattern:

`update(Key, NewV), item(Key, OldV) <=> item(Key, NewV) .`

- ▶ e.g. `upd25(A, X, Y), c(A, B, C, D, E) <=> c(A, X, C, D, Y) .`



## Suspension reuse

20/37

- ▶ 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.
```

- ▶  $\rightsquigarrow$  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

20/37

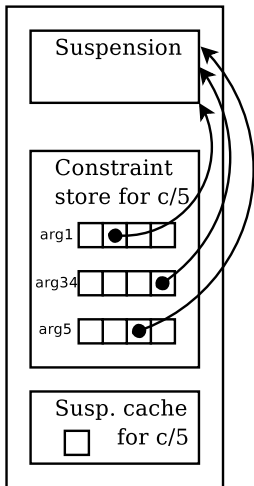
- ▶ Pattern of in-place updates is often used...  
...but: many other ways to remove/insert a constraint
- ▶ For example:

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

- ▶  $\rightsquigarrow$  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

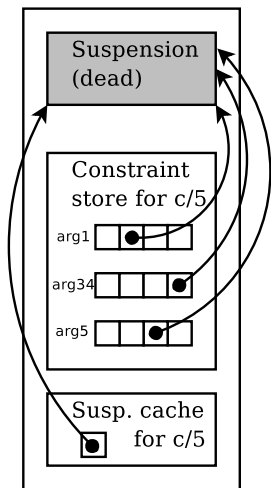
21/37



- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

# Suspension reuse

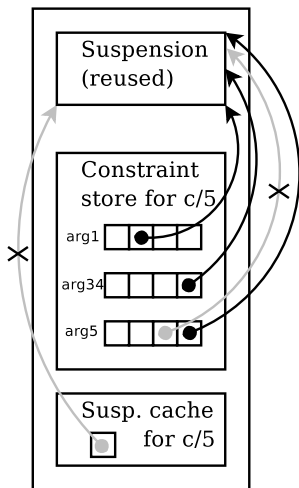
21/37



- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

# Suspension reuse

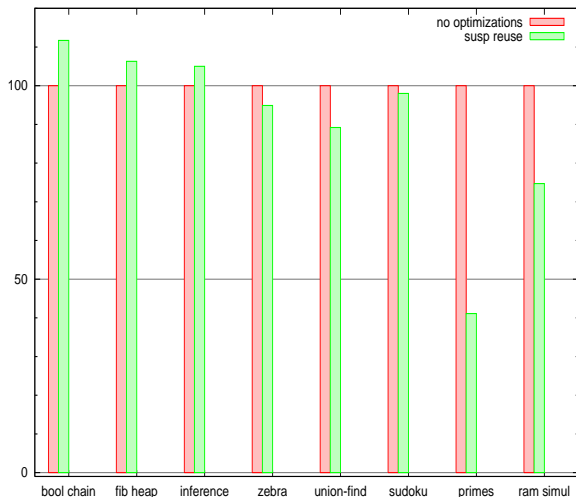
21/37



- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

# Time Results

22/37

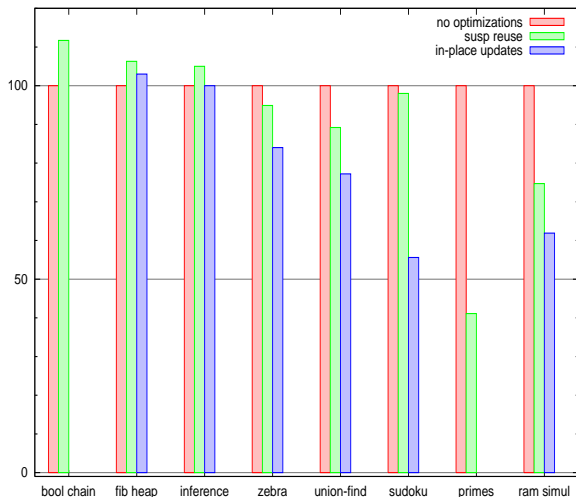


## TIME:

- ▶ 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

22/37

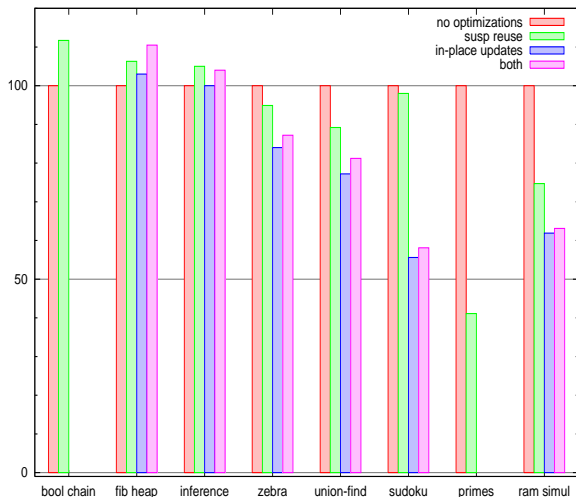


## TIME:

- ▶ 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

22/37

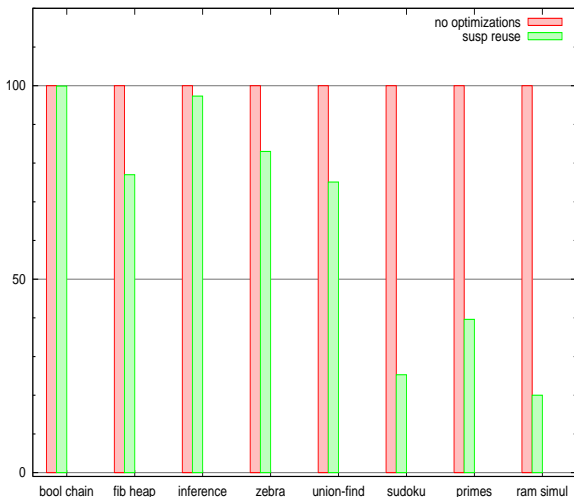


## TIME:

- ▶ 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

23/37

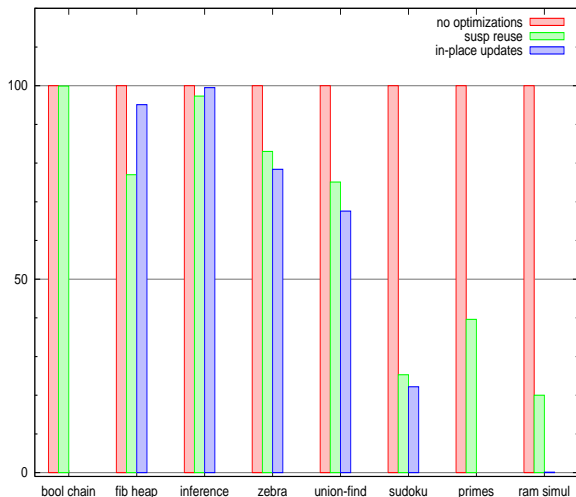


## SPACE:

- ▶ 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

23/37

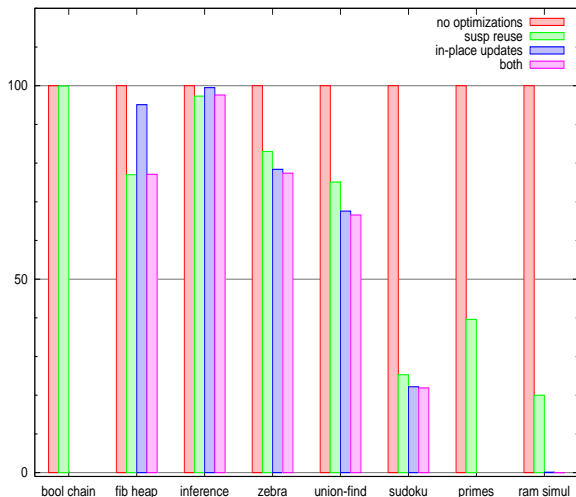


## SPACE:

- ▶ 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)

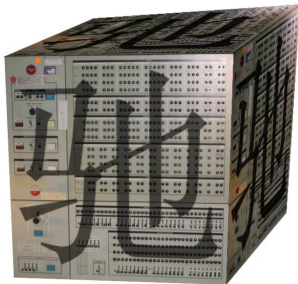
# Space Results

23/37



## 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)



- 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

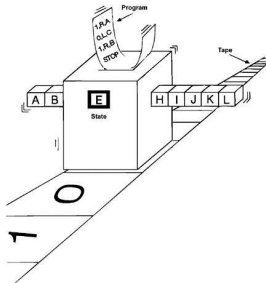
# Computational power and complexity of CHR

25/37

- ▶ Paper at CHR'05 workshop; paper submitted to TOPLAS
- ▶ *CHR machine* vs. TM and RAM



CHR machine



Turing machine



Random Access Memory machine

## Complexity of CHR

26/37

- ▶ 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

26/37

- ▶ 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

26/37

- ▶ 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

26/37

- ▶ 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

26/37

- ▶ 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**

## RAM simulator in CHR

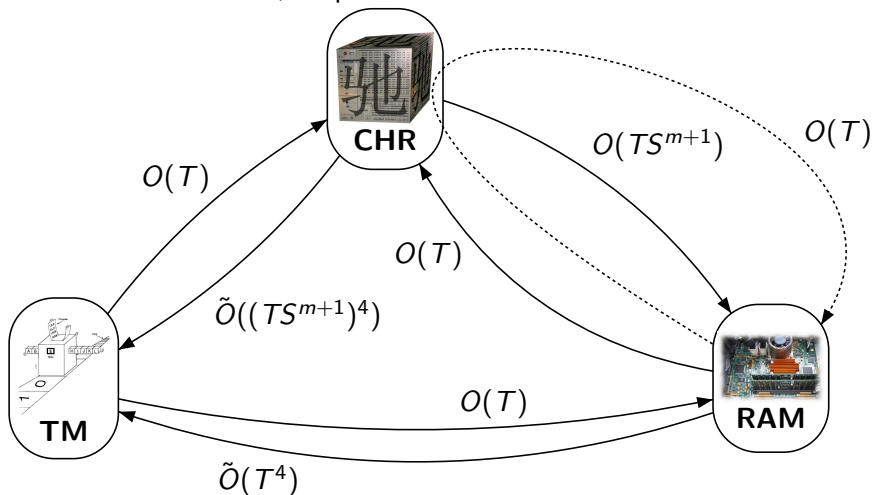
27/37

**m**/*2* : memory cells (address,value) **pc**/*1* : program counter (label)  
**prog**/*{2,3,4}* : program instructions (label,instruction,arguments)

```
prog(L,init,A), m(A,B) \ pc(L) <=> m(B,0), pc(L+1).
prog(L,cnst,B,A) \ m(A,X), pc(L) <=> m(A,B), pc(L+1).
prog(L,add,B,A), m(B,Y) \ m(A,X), pc(L) <=> m(A,X+Y), pc(L+1).
prog(L,sub,B,A), m(B,Y) \ m(A,X), pc(L) <=> m(A,X-Y), pc(L+1).
prog(L,mul,B,A), m(B,Y) \ m(A,X), pc(L) <=> m(A,X*Y), pc(L+1).
prog(L,div,B,A), m(B,Y) \ m(A,X), pc(L) <=> m(A,X/Y), pc(L+1).
prog(L,mov,B,A), m(B,Y) \ m(A,X), pc(L) <=> m(A,Y), pc(L+1).
prog(L,imv,B,A), m(B,C), m(C,Y) \ m(A,X), pc(L) <=> m(A,Y), pc(L+1).
prog(L,mvi,B,A), m(B,Y), m(A,C) \ m(C,X), pc(L) <=> m(C,Y), pc(L+1).
prog(L,jmp,A) \ pc(L) <=> pc(A).
prog(L,cjmp,A,J), m(A,0) \ pc(L) <=> pc(J).
prog(L,cjmp,A,J), m(A,X) \ pc(L) <=> X ≠ 0 | pc(L+1).
prog(L,halt) \ pc(L) <=> true.
```

# Complexity results

$A \xrightarrow{X} B$  : a  $T$ -time,  $S$ -space  $A$  can be simulated on a  $X$ -time  $B$ .



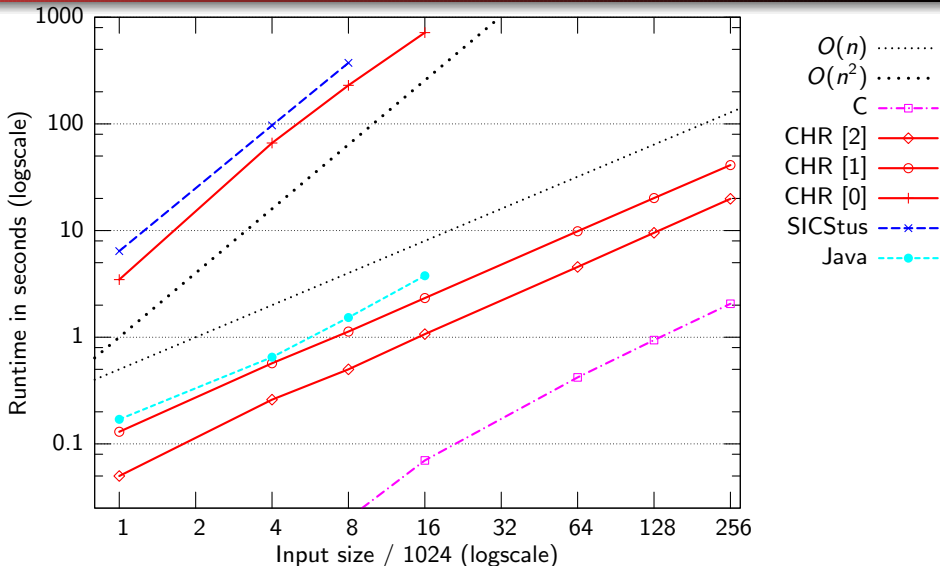
## Constant factors

29/37

- ▶ 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

30/37



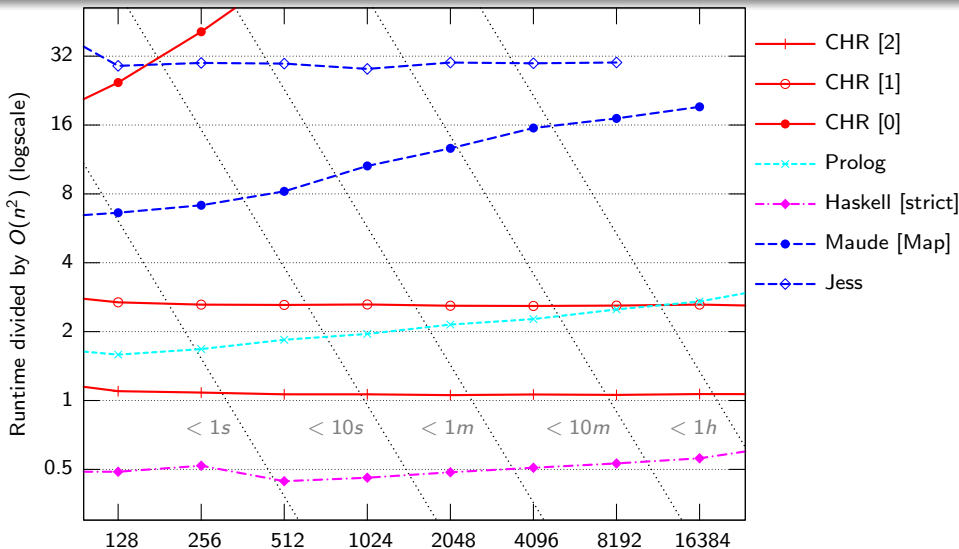
## Other declarative languages

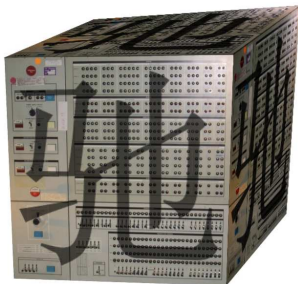
31/37

- ▶ 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]

32/37





- 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<sup>¬</sup>: CHR with negation as absence

34/37

- ▶ 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).`
- ▶ Without negated heads also possible in refined semantics:  
`person(X) ==> check_married(X).`  
`married(X,_) \ check_married(X) <=> true.`  
`check_married(X) <=> single(X).`
- ▶ CHR<sup>¬</sup> allows shorter, more readable programs
- ▶ CHR<sup>¬</sup> also triggers rules on removal of negated heads (harder to do manually)
- ▶ However, programming in CHR<sup>¬</sup> seems to be tricky

## Aggregates in CHR

35/37

- ▶ Ongoing work, maybe CHR'07
- ▶ Add aggregates to CHR: count, sum, max, avg, ...
- ▶ Negation is special case of this:  
 $\text{not}(\text{NH}) \equiv \text{count}(\text{NH}, 0)$
- ▶ Also user-defined aggregates, nested aggregates
- ▶ Example: a graph is Eulerian if  $\forall$  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 with aggregates

36/37

Dijkstra's algorithm in CHR:

```
:- 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 with aggregates

36/37

Dijkstra's algorithm in CHR with aggregates:

```
:- 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), no(dist(B,_)) ==> newlabel(B,D+W).  
  
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 with aggregates

36/37

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?*