

# OPTIMIZING COMPILATION AND COMPUTATIONAL COMPLEXITY OF CONSTRAINT HANDLING RULES

Jon Sneyers  
K.U.Leuven, Belgium

*Preliminary Ph.D. Defense, October 2008*



## Evolution of programming languages:

- ▶ 1950s: **assembly** languages *very low-level*
- ▶ 1960s: **imperative** languages ↓
- ▶ 1970s: LP, **declarative** languages ↓
- ▶ 1980s: CLP ↓
  - ▶ “black-box” CLP systems ↓
  - ▶ “glass-box” CLP systems ↓
- ▶ 1990s: CHR *very high-level*

# Constraint Handling Rules [Frühwirth 1991]

2/26

- ▶ Very high-level programming language extension
- ▶ Concurrent multi-headed committed-choice guarded rules
- ▶ Growing scope:
  - ▶ Originally: special-purpose for implementing constraint solvers
  - ▶ Later: reasoning systems in general
  - ▶ Today: general-purpose language

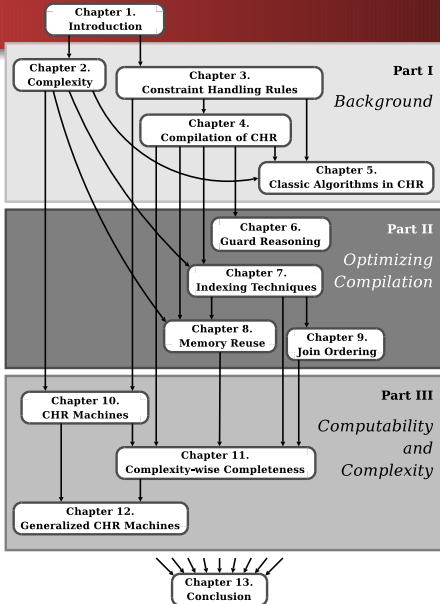
# Goals

3/26

- ▶ Show/enhance usability of CHR as general-purpose language
  - ▶ Study performance of CHR
    - ▶ complexity of CHR language
    - ▶ complexity of CHR systems
  - ▶ Improve performance of CHR
    - ▶ new compiler optimizations
  - ▶ Time and space
  - ▶ Asymptotic complexity and constant factors

## Overview

4/26

**1** Background

- 2. Complexity
- 3. Constraint Handling Rules
- 4. Compilation of CHR
- 5. Classic Algorithms in CHR

**2** Optimizing Compilation of CHR

- 6. Guard Reasoning
- 7. Indexing Techniques
- 8. Memory Reuse
- 9. Join Ordering

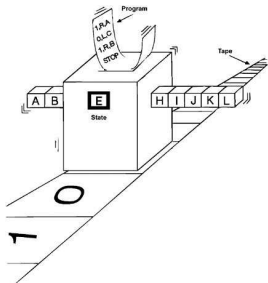
**3** Computability and Complexity of CHR

- 10. CHR Machines
- 11. Complexity-wise Completeness
- 12. Generalized CHR Machines
- 13. Conclusion

# Complexity theory

5/26

Models of computation:



Turing machine



Random Access Memory machine

## Syntax and semantics of CHR on 1 slide

6/26

- ▶ **CHR( $\mathcal{H}$ )** where  $\mathcal{H}$  is host-language
  - ▶ CHR constraints, defined in CHR program
  - ▶ Built-in (host-language) constraints, theory  $\mathcal{D}_{\mathcal{H}}$
- ▶ **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:**
  - ▶ 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

6/26

- ▶ **CHR( $\mathcal{H}$ )** where  $\mathcal{H}$  is host-language
  - ▶ CHR constraints, defined in CHR program
  - ▶ Built-in (host-language) constraints, theory  $\mathcal{D}_{\mathcal{H}}$
- ▶ **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:**
  - ▶ 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

6/26

- ▶ **CHR( $\mathcal{H}$ )** where  $\mathcal{H}$  is host-language e.g.  $\mathcal{H} = \text{Prolog}$ 
  - ▶ **CHR constraints**, defined in CHR program e.g.  $\leq$
  - ▶ **Built-in (host-language) constraints**, theory  $\mathcal{D}_{\mathcal{H}}$  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:**
  - ▶ 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

6/26

- ▶ **CHR( $\mathcal{H}$ )** where  $\mathcal{H}$  is host-language e.g.  $\mathcal{H} = \text{Prolog}$ 
  - ▶ CHR constraints, defined in CHR program e.g.  $\leq$
  - ▶ Built-in (host-language) constraints, theory  $\mathcal{D}_{\mathcal{H}}$  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:**
  - ▶ 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  $h_2$

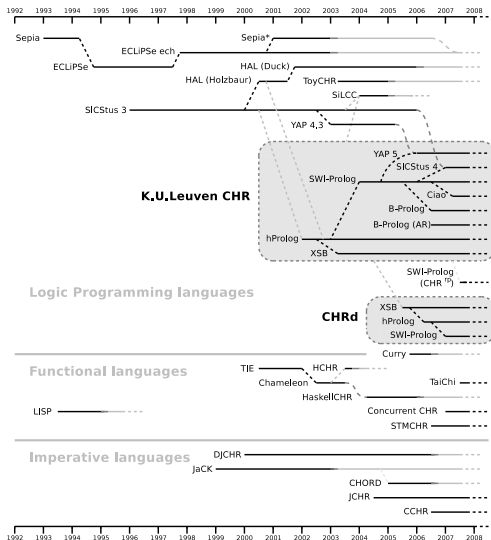
## Syntax and semantics of CHR on 1 slide

6/26

- ▶ **CHR( $\mathcal{H}$ )** where  $\mathcal{H}$  is host-language e.g.  $\mathcal{H} = \text{Prolog}$ 
  - ▶ CHR constraints, defined in CHR program e.g.  $\leq$
  - ▶ Built-in (host-language) constraints, theory  $\mathcal{D}_{\mathcal{H}}$  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:**
  - ▶ 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 systems

7/26



## Recent trends:

- ▶ Many new CHR(Prolog) systems (every self-respecting Prolog has CHR now)
- ▶ Concurrent CHR(Haskell) systems
- ▶ Fast systems in Java & C

## Other topics discussed in Chapter 3

8/26

- ▶ Program properties: a.o. confluence, termination
- ▶ Extensions of CHR
  - ▶ disjunction & search
  - ▶ negation-as-absence & aggregates
  - ▶ adaptive CHR
- ▶ Example CHR programs
  - ▶ Fibonacci numbers, Zebra puzzle, Sudoku
- ▶ Applications of CHR
- ▶ Related formalisms
  - ▶ Join-Calculus
  - ▶ Logical Algorithms
  - ▶ Graph Transformation Systems
  - ▶ Petri nets

# CHR compilation

9/26

- ▶ Standard compilation scheme [Holzbaur & Frühwirth 1998]
- ▶ Formalized as the *refined semantics*  $\omega_r$  [Duck+ 2004]
- ▶ Instantiates the abstract operational semantics:
  - ▶ Conjunctions: depth-first, left-to-right (as in Prolog)
  - ▶ Textual order rule application, notion of *active constraint*

# Implementing classic algorithms in CHR

5.3 Hopcroft's algorithm for finite automata

87

- Step 1.**  $\forall s \in S, a \in I$ , construct  $\delta^{-1}(s, a) = \{t \mid \delta(t, a) = s\}$ .
- Step 2.** Construct  $B(1) = F$ ,  $B(2) = S \setminus F$ , and for each  $a \in I$  and  $i \in \{1, 2\}$ , construct  $A(a, i) = \{s \mid s \in B(i) \text{ and } \delta^{-1}(s, a) \neq \emptyset\}$ .
- Step 3.** Set  $k = 3$ .
- Step 4.**  $\forall a \in I$  construct  $L(a) = \begin{cases} \{1\} & \text{if } |A(a, 1)| \leq |A(a, 2)| \\ \{2\} & \text{otherwise} \end{cases}$
- Step 5.** Select  $a$  in  $I$  and  $i$  in  $L(a)$ . Stop if  $L(a) = \emptyset$  for each  $a$  in  $I$ .
- Step 6.** Delete  $i$  from  $L(a)$ .
- Step 7.**  $\forall j < k$  such that  $\exists t \in B(j) : \delta(t, a) \in A(a, i)$ , do the following:
- 7a.** Partition  $B(j)$  into  $B'(j) = \{t \mid \delta(t, a) \in A(a, i)\}$  and  $B(j) \setminus B'(j)$ .
  - 7b.** Construct  $B(k) = B(j) \setminus B'(j)$  and replace  $B(j)$  by  $B'(j)$ . Construct corresponding  $A(a, j)$  and  $A(a, k)$  for each  $a$  in  $I$ .
  - 7c.**  $\forall a \in I$  set  $L(a) = \begin{cases} L(a) \cup \{j\} & \text{if } j \notin L(a) \text{ and } 0 < |A(a, j)| \leq |A(a, k)| \\ L(a) \cup \{k\} & \text{otherwise} \end{cases}$
  - 7d.** Set  $k = k + 1$ .
- Step 8.** Return to Step 5.

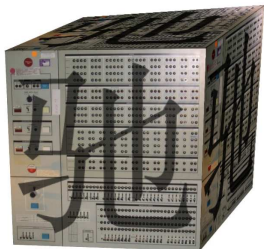
Figure 5.1: Pseudo-code description of Hopcroft's algorithm

Listing 5.5: HOPCROFT: Hopcroft's algorithm for minimizing finite automata

```

init, final(S) \ state(S) <=> b(1,S). % step 2
init \ state(S) <=> b(2,S).
b(1,S), input(A) => nb_a(A,1,0), add_a(A,1,S).
add_a(A,1,T) \ a(A,J,T) <=> nb_a(A,J,-1).
delta(_,A,S) \ add_a(A,I,S) <=> nb_a(A,I,1), a(A,I,S).
add_a(_____) <=> true.
nb_a(A,I,X), nb_a(A,I,Y) <=> Z is X+Y, nb_a(A,I,Z).
init <=> k(3), add_to_1(1,2), main_loop. % steps 3 and 4
add_to_1(J,K), input(A), nb_a(A,J,X), nb_a(A,K,Y)
=> (X <= Y -> 1(A,J) ; 1(A,K)).
add_to_1(____) <=> true.
main_loop, 1(A,I) <=> part(A,I). % steps 5 and 6
main_loop <=> true. % all L's empty: stop
part(A,I), a(A,I,X), delta(T,A,X) \ b(J,T) <=> bp(J,T). % step 7
part(____), bp(J,_)#passive \ k(K) <=> do_part(J,K).
do_part(J,K) \ bp(J,T)#passive <=> b(K,T).
do_part(J,K) <=> add_to_1(K,J), K1 is K+1, k(K1).
part(____) <=> main_loop. % step 8
    
```

- ▶ Union-find [Schrijvers & Frühwirth 2006]
- ▶ Dijkstra's shortest path algorithm
  - ▶ with Fibonacci-heaps as priority queue
- ▶ Hopcroft's algorithm for minimizing finite automata
- ▶ CHR as efficient 'executable pseudo-code'



- 1 Background
  - 2. Complexity
  - 3. Constraint Handling Rules
  - 4. Compilation of CHR
  - 5. Classic Algorithms in CHR
- 2 **Optimizing Compilation of CHR**
  - 6. Guard Reasoning
  - 7. Indexing Techniques
  - 8. Memory Reuse
  - 9. Join Ordering
- 3 **Computability and Complexity of CHR**
  - 10. CHR Machines
  - 11. Complexity-wise Completeness
  - 12. Generalized CHR Machines
  - 13. Conclusion

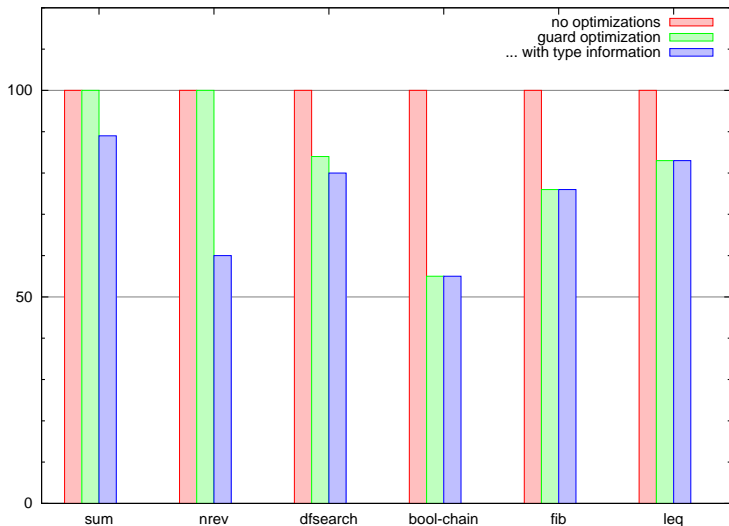
## Guard reasoning

12/26

- ▶ **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

## Effect on runtime

13/26



# Indexing techniques

14/26

How to implement constraint store lookups?

- ▶ Naive approach: one big list
- ▶ Attributed variables
- ▶ Hash tables
  - ▶ Ground hash tables
  - ▶ Non-ground hash tables
- ▶ Arrays (`dense_int`)

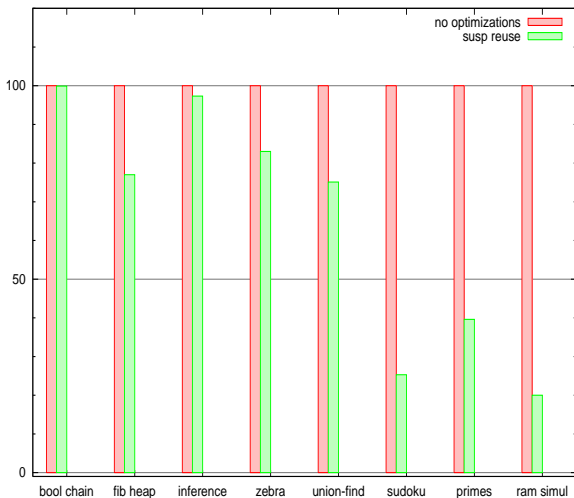
# Memory reuse

15/26

- ▶ Suspension reuse & in-place updates ▶ Details
- ▶ “*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

# Space Results

16/26



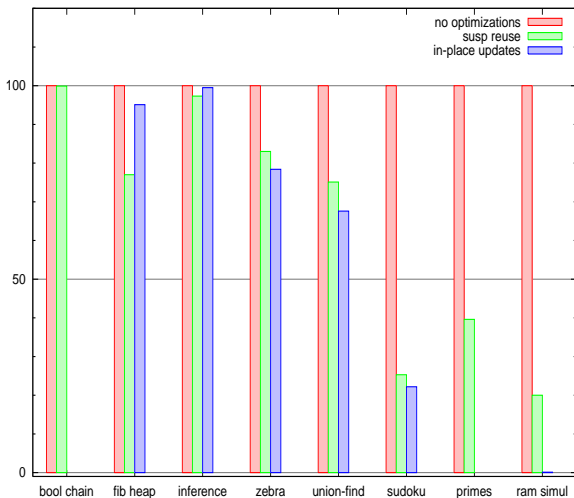
## SPACE:

- ▶ 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)

▶ Time results

# Space Results

16/26



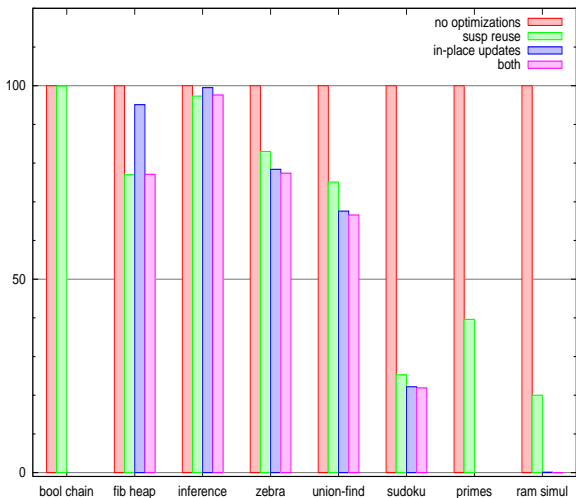
## SPACE:

- ▶ 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)

▶ Time results

# Space Results

16/26



## 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)

▶ Time results

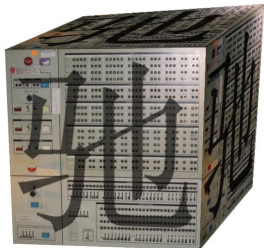
## The join ordering problem

17/26

- ▶ *Join*: finding matching partners for a given active occurrence
  - ▶ common approach: nested loops
  - ▶ using indexes to take equality guards into account
- ▶ *Ordering*: finding a (good) order in which to do the lookups
- ▶ E.g. consider the active occurrence  $\text{part}/2$  in the rule  
 $\text{part}(A,I), \text{delta}(T,A,X), a(A,I,X) \setminus b(J,T) \Leftrightarrow b'(J,T)$ .  
 3 partner constraints, so  $3! = 6$  possible join orders:

<pre>foreach(delta(T,A,X)) {   foreach(a(A,I,X)) {     foreach(b(J,T)) {       call(b'(J,T))     }   } }</pre>	<pre>foreach(a(A,I,X)) {   foreach(delta(T,A,X)) {     foreach(b(J,T)) {       call(b'(J,T))     }   } }</pre>	<pre>foreach(b(J,T)) {   foreach(delta(T,A,X)) {     foreach(a(A,I,X)) {       call(b'(J,T))     }   } }</pre>	...
--	--	--	-----

▶ Details



- 1 Background
  - 2. Complexity
  - 3. Constraint Handling Rules
  - 4. Compilation of CHR
  - 5. Classic Algorithms in CHR
- 2 Optimizing Compilation of CHR
  - 6. Guard Reasoning
  - 7. Indexing Techniques
  - 8. Memory Reuse
  - 9. Join Ordering
- 3 Computability and Complexity of CHR
  - 10. CHR Machines
  - 11. Complexity-wise Completeness
  - 12. Generalized CHR Machines
  - 13. Conclusion

## Computability of CHR

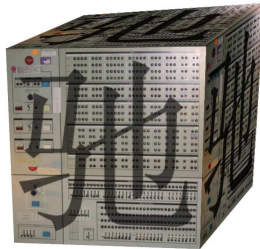
19/26

- ▶ CHR (without host language) is Turing-complete
- ▶ Turing-complete subsets of CHR

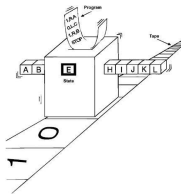
		only one rule				multiple rules			
		single-headed		multi-headed		single-headed		multi-headed	
		P	S	P	S	P	S	P	S
propositional CHR (no arguments)	abstract							<i>Theorem 10.5</i>	
	refined			(a)	?			(a)	MINSKY -P
	CHR-only	<i>Cor. 10.6</i>		?	TMSIM -1R	<i>Cor. 10.6</i>		TMSIM -PROP	TMSIM
	CHR with sufficiently strong host language	?	?	?		(b)	MINSKY -A		
	CHR with Turing-complete host language								

# CHR machines

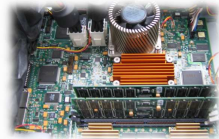
20/26



CHR machine



Turing machine



RAM machine

# Complexity of CHR

21/26

- ▶ 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

# Complexity of CHR

21/26

- ▶ 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 simulator
- ▶ 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

21/26

- ▶ 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 simulator
- ▶ 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

21/26

- ▶ 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 simulator
- ▶ 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

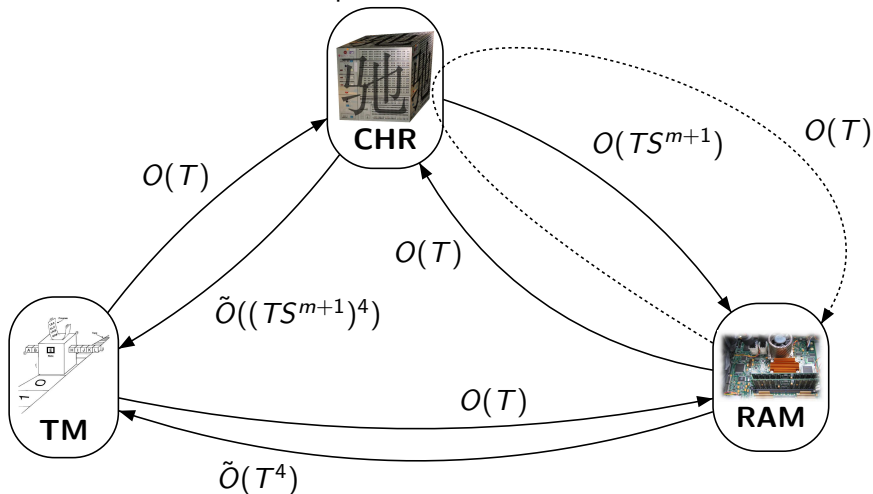
21/26

- ▶ 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 simulator
- ▶ 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 results

22/26

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



## Other declarative languages

23/26

- ▶ 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 results](#)

# Constant factors

24/26

- ▶ Compare CHR implementation with C implementation
- ▶ Two case studies:
  - ▶ Union-find
  - ▶ Dijkstra's algorithm with Fibonacci-heaps
- ▶ C is “only” 10 times faster
- ▶ Space:
  - ▶ factor 10 for union-find
  - ▶ factor 3 for Dijkstra's algorithm

[▶ Benchmark results](#)

# Generalized CHR machines

25/26

- ▶ Execution strategies, strategy class
- ▶ Generalized confluence
- ▶ General CHR machines
- ▶ Non-deterministic CHR machines
- ▶ CHR machines with Stored Program

▶ Details

▶ Details

▶ Details

▶ Details

# Conclusion

26/26

- ▶ Future work:
  - ▶ Programming environments & tools
  - ▶ Execution control (cf. Leslie's thesis)
  - ▶ Parallelism & concurrency
  - ▶ Scalability
    - ▶ huge programs, huge constraint stores
    - ▶ incremental compilation, run-time rule assertion
    - ▶ modularity, standardized libraries
    - ▶ ...

▶ *Questions?*

# Conclusion

26/26

- ▶ Future work:
  - ▶ Programming environments & tools
  - ▶ Execution control (cf. Leslie's thesis)
  - ▶ Parallelism & concurrency
  - ▶ Scalability
    - ▶ huge programs, huge constraint stores
    - ▶ incremental compilation, run-time rule assertion
    - ▶ modularity, standardized libraries
    - ▶ ...

▶ *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([],S) <=> S=0.`  
`sum([X|Xs],S) <=> sum(Xs,T), S is X+T.`
- ▶  $\Rightarrow$  `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:  
`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!

- ▶ 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!

- ▶ 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!

- ▶ 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!  $\Rightarrow$  cleaner generated code

```

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

```

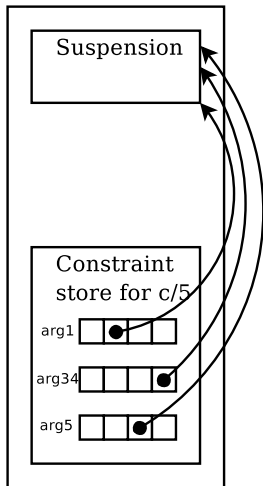
:- use_module(library(chr_runtime)).
:- use_module(library(chr_hashtable_store)).
'$enumerate_suspensions' (_) :- fail.
sum([],A) :- !, A=0.
sum([D|C],B) :- sum(C,A), B is D+A.

```

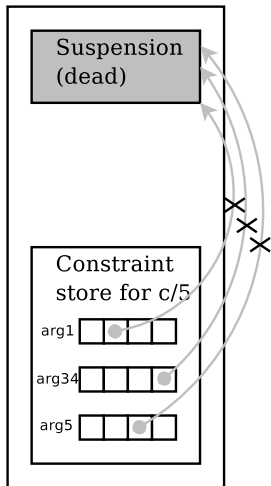
- ▶ Suspension reuse & in-place updates ◀ Return
- ▶ “*Suspension*”: internal representation of CHR constraints as a Prolog term: e.g.  $A \leq B$  could be represented as a term  $S = \text{suspension}(37, \text{stored}, \text{'}\leq/2\_0\text{'}(A, B, S), \text{nohist}, \leq, A, B)$
- ▶ Constraint 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

- ▶ Suspension reuse & in-place updates ◀ Return
- ▶ “*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

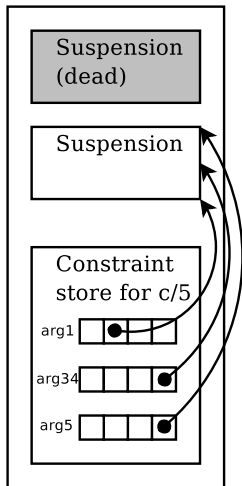
- ▶ Suspension reuse & in-place updates ◀ Return
- ▶ “*Suspension*”: internal representation of CHR constraints as a Prolog term: e.g.  $A \leq B$  could be represented as a term  $S = \text{suspension}(37, \text{stored}, \text{'}\leq/2\_0\text{'}(A, B, S), \text{nohist}, \leq, A, B)$
- ▶ Constraint 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



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



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

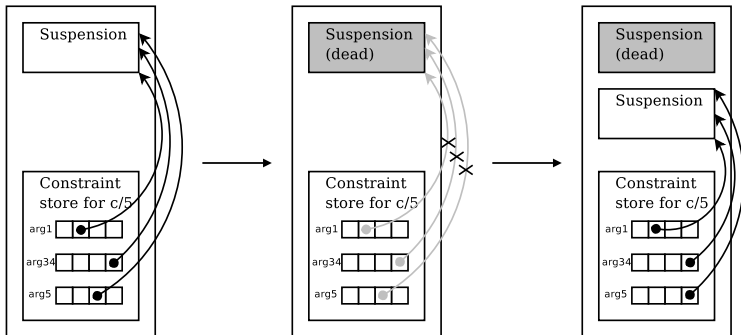


- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

- ▶ 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) .



- ▶ 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) .



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

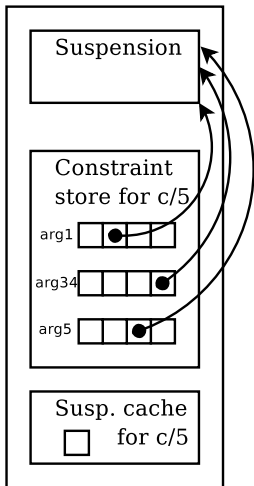
```
candidate(1) <=> true.  
candidate(N) <=> N>1 | prime(N), candidate(N-1).  
prime(I) \ prime(J) <=> J mod I =:= 0 | true.
```

- ▶  $\rightsquigarrow$  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

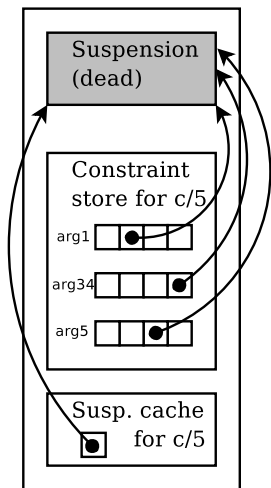
- ▶ Pattern of in-place updates is often used...  
...but: many other ways to remove/insert a constraint
- ▶ For example:

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

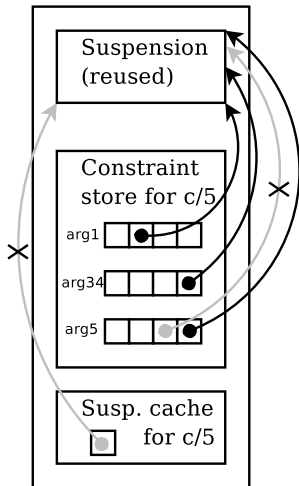
- ▶  $\rightsquigarrow$  Suspension reuse = dynamic in-place updates
- ▶ Reuse of an old idea:
  - ▶ maintain a cache of old suspensions
  - ▶ when constraint is removed, put suspension in cache and keep constraint in data structures (but mark it)
  - ▶ when a suspension is created, first check cache
  - ▶ dynamically check which data structures to fix



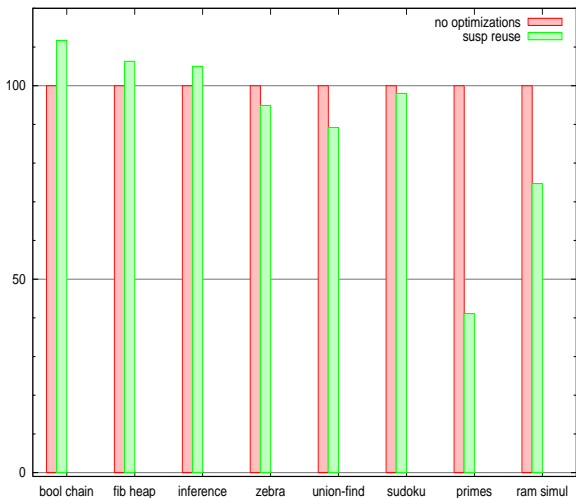
- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion



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

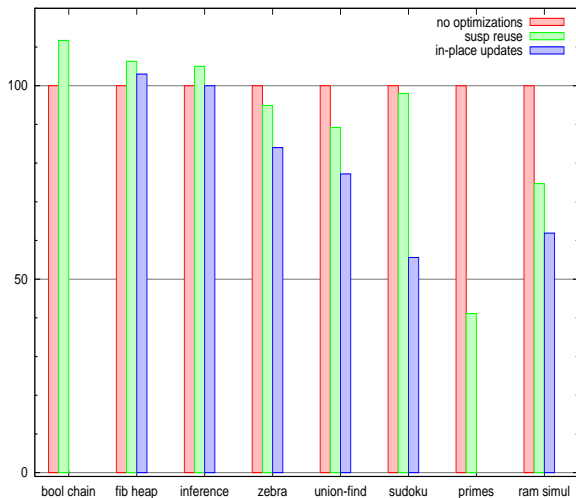


- ▶ Constraint insertion
- ▶ Constraint removal
- ▶ New constraint insertion

**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

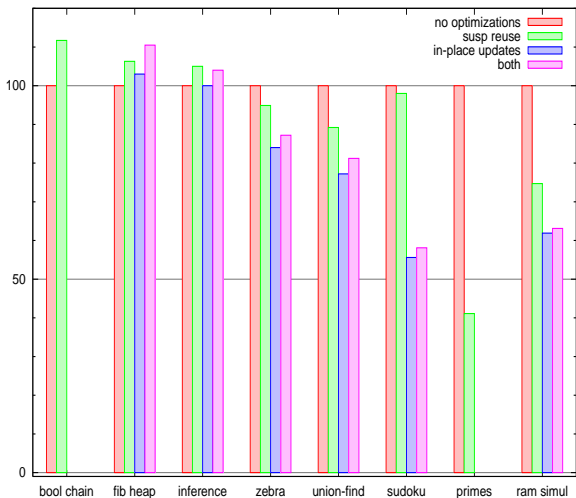
[◀ Return](#)



## 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

[Return](#)



## 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

[◀ Return](#)

- ▶ Early CHR systems allowed only single and two-headed rules  
⇒ at most 1 partner constraint, so nothing to order
- ▶ Hence, old CHR programs (e.g. `leq`) use at most 2 heads
- ▶ More recent programs have more heads:

Name	1	2	3	4	5	6	Total
EU Car Rental	5	4	2	5	2		18
Hopcroft	6	7	2	2			17
Monkey & Bananas	1	7	15	2			25
RAM Simulator	1	2	3	5	2		13
Timed Automaton		11	10	3			24
Type Inference	26	48	13	6	4		97
Well-founded Semantics	3	25	8	4	1	2	43
Total	42	104	53	27	9	2	237

- ▶ Why care about join order?  
*Wrong join order often means wrong time complexity!*
- ▶ Why not just leave it to the programmer? (and simply use textual order)
  - ▶ *Programmer should not worry about “low-level” details*
  - ▶ *Multi-headed rules may have many active occurrences, so there is not always a way to arrange the heads such that textual order is optimal*
  - ▶ *Sometimes the optimal order depends on dynamic properties of the store*

- ▶ Current implementations do static join ordering based on ad-hoc heuristics
- ▶ We propose a more precise cost model
- ▶ We also consider dynamic join ordering
- ▶ Typical information/time conflict:
  - ▶ Statically (at compile time) we can spend much time on finding the optimal order, but we have little information
  - ▶ Dynamically (at runtime) we have all information, but no time
- ▶ However, in some cases there is no single optimal join order, so dynamic ordering is needed to obtain correct complexity

$m/2$  : memory cells (address,value)    $c/1$  : program counter (label)  
 $i/\{2,3,4\}$  : program instructions (label,instruction,arguments)

## RAMSIMUL

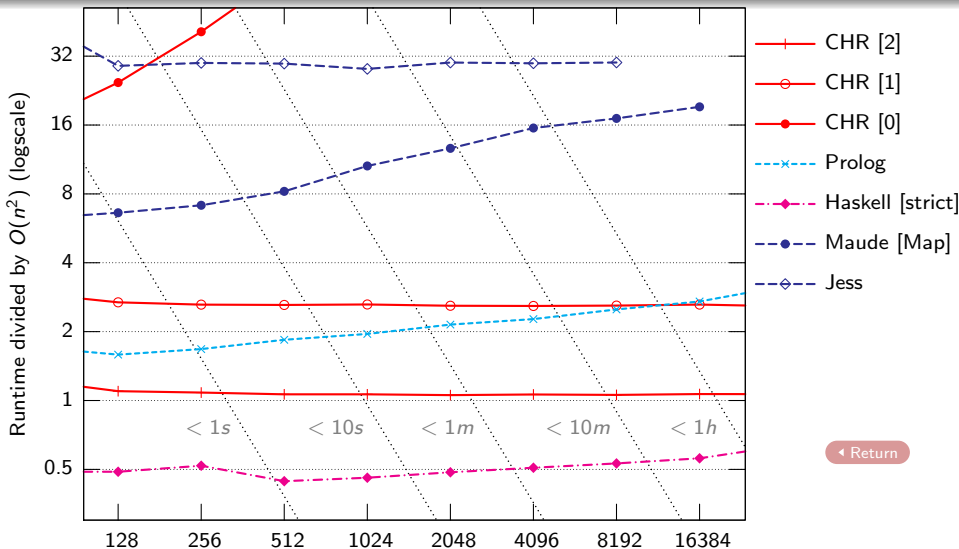
```

i(L,init,A), m(A,B), maxm(M) \ c(L) <=> initm(M+1,B,L).
  initm(A,B,L) <=> A <= B | m(A,0), initm(A+1,B,L).
  initm(A,B,L), m(B,X) <=> A > B | m(B,0), maxm(B), c(L+1).
i(L,cnst,B,A) \ m(A,X), c(L) <=> m(A,B), c(L+1).
i(L,add,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X+Y), c(L+1).
i(L,sub,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X-Y), c(L+1).
i(L,mul,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X*Y), c(L+1).
i(L,div,B,A), m(B,Y) \ m(A,X), c(L) <=> m(A,X//Y), c(L+1).
i(L,mov,B,A), m(B,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,imv,B,A), m(B,C), m(C,Y) \ m(A,_), c(L) <=> m(A,Y), c(L+1).
i(L,mvi,B,A), m(B,Y), m(A,C) \ m(C,_), c(L) <=> m(C,Y), c(L+1).
i(L,jmp,A) \ c(L) <=> c(A).
i(L,cjmp,A,J), m(A,0) \ c(L) <=> c(J).
i(L,cjmp,A,J), m(A,X) \ c(L) <=> X \= 0 | c(L+1).
i(L,halt) \ c(L) <=> true.

```

# Benchmark: RAM simulator [nested loop]

40/26



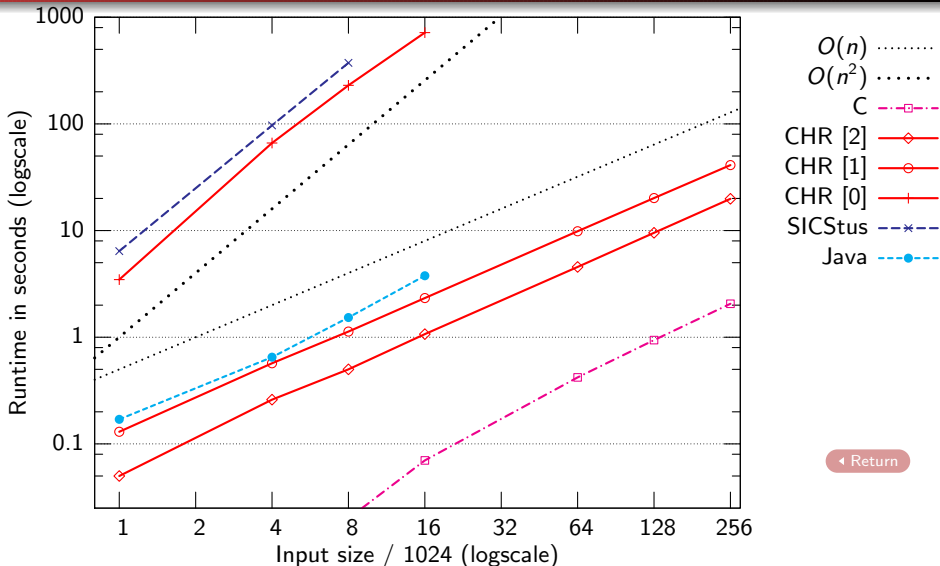
Return

<i>Language</i>	<i>Time</i>	<i>Space</i>	<i>Complexity-wise complete?</i>
CHR [0]	not optimal	optimal	space-only
CHR [1]	optimal	optimal	yes
CHR [2]	optimal	optimal	yes
Prolog	almost optimal	not optimal	almost (time-only)
Haskell (strict)	almost optimal	not optimal	almost (time-only)
Haskell (lazy)	not optimal	not optimal	no
Maude (naive)	not optimal	optimal	space-only
Maude (Map)	almost optimal	optimal	almost
Jess	optimal	optimal	yes

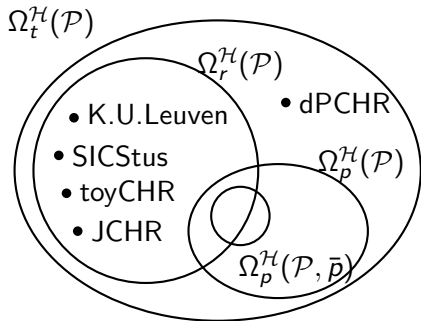
[← Return](#)

# Benchmark: Dijkstra's shortest path algorithm

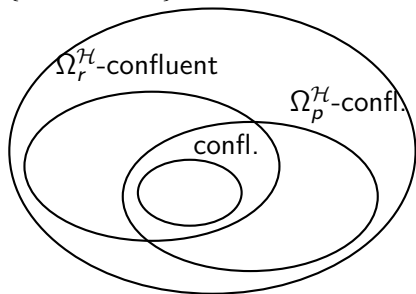
42/26



Return

*Execution strategies*

{K.U.Leuven}-confluent programs

*CHR programs*

← Return

$$P = P_{\Omega_t^{\mathcal{H}}} \subseteq P_{\Omega_r^{\mathcal{H}}} \subseteq P_{\{\text{K.U.Leuven}\}} = NP_{\{\text{K.U.Leuven}\}} \subseteq NP_{\Omega_r^{\mathcal{H}}} \subseteq NP_{\Omega_t^{\mathcal{H}}} = NP$$

- ▶ most of the inclusions collapse to equalities:
  - ▶  $P_{\Omega_t^{\mathcal{H}}} = P_{\{\text{K.U.Leuven}\}}$  because the program RAMSIMUL is  $(\Omega_t^{\mathcal{H}}\text{-})$ confluent

- ▶ Consider NCHR program 3SAT:

```
clause(A,_,_) <=> true(A).  
clause(_,B,_) <=> true(B).  
clause(_,_,C) <=> true(C).  
true(X), true(not(X)) <=> fail.
```

- ▶ Corresponding NCHR machine  $(\emptyset, \Omega_t^{\mathcal{H}}, 3SAT, 3SATCLAUSES)$  decides 3SAT in linear time
- ▶ 3SAT is NP-complete, so  $NP \subseteq NP_{\Omega_t^{\mathcal{H}}}$

- ▶ encode CHR program using “reserved keyword” constraints
- ▶ rule/1, khead/2, rhead/2, guard/2, body/2
- ▶ Example:

```
foo @ bar ==> baz.  
qux @ blarg \ wibble <=> flob | wobble.
```

would be encoded as

```
rule(foo), khead(foo,bar), guard(foo,true), body(foo,baz),  
rule(qux), khead(qux,blarg), rhead(qux,wibble),  
                    guard(qux,flob), body(qux,wobble)
```

- ▶ program in constraint store
- ▶ program is put in the query (or in the initial store)
- ▶ rule can only fire if there are no pending rule components to be **Introduced**
  
- ▶ CHRSP machine decides co-NP-complete languages in only linear time
- ▶ Example: Hamiltonian paths in directed graphs

- ▶ language of graphs without Hamiltonian path is co-NP-complete
- ▶ consider this CHRSP program:

```
size(N) <=> rule(find_path), size(N,A).
size(N,A) <=> N>1 |
    khead(find_path,node(A)), khead(find_path,edge(A,B)),
    size(N-1,B).
size(1,A) <=>
    khead(find_path,node(A)), body(find_path,fail).
```

- ▶ input constraints:  $\text{edge}/2, \text{node}/1, \text{size}(n)$  ( $n = \# \text{nodes}$ )
- ▶ program makes rule: `find_path @ node(A1),node(A2),...,node(An), edge(A1,A2),edge(A2,A3),...,edge(An-1,An) ==> fail.`
- ▶ this rule fires (rejecting the input graph) if and only if the graph has a Hamiltonian path