Constraint Handling Rules
A Tutorial for (Prolog) Programmers

Tom Schrijvers

Katholieke Universiteit Leuven, Belgium
Tom.Schrijvers@cs.kuleuven.be


DTAI
Introduction
The Art of CHR
Simpagation
For the Sake of Arguments
En Guarde
A Perfect Match
Bigger Programs
Propagation
Order in the Rules
Reactivation
CHR vs. Prolog
Tutorial Summary
Facts about CHR
About This Tutorial

Expected background:

- know & like programming
Expected background:

- know & like programming
- know Prolog, ...
Expected background:

- know & like programming
- know Prolog, ...
- but have an itch that Prolog doesn’t scratch
Expected background:

- know & like programming
- know Prolog, . . .
- but have an itch that Prolog doesn’t scratch
- . . . or you will have at the end of this tutorial
Expected mindset:

- forget about Prolog for now
Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
- really, they won’t be the same!!!
Expected mindset:

▶ forget about Prolog for now
▶ do not assume things to work as in Prolog
▶ really, they won’t be the same!!!

You’ve been warned!
About This Tutorial

Expected mindset:

- forget about Prolog for now
- do not assume things to work as in Prolog
- really, they won’t be the same!!!

You’ve been warned! Also:

- do not expect anything from the word “constraint”
About This Tutorial

Expected mindset:

▶ forget about Prolog for now
▶ do not assume things to work as in Prolog
▶ really, they won’t be the same!!!

You’ve been warned! Also:

▶ do not expect anything from the word “constraint”
▶ CHR constraints aren’t
What you’ll learn:
About this tutorial

What you’ll learn:

1. how CHR works (operationally)
About this tutorial

What you’ll learn:

1. how CHR works (operationally)
2. how to program in CHR
What you’ll learn:

1. how CHR works (operationally)
2. how to program in CHR
3. just maybe, what CHR is good for
1. Introduction
2. The Art of CHR
3. Simpagation
4. For the Sake of Arguments
5. En Guarde
6. A Perfect Match
7. Bigger Programs
8. Propagation
9. Order in the Rules
10. Reactivation
11. CHR vs. Prolog
12. Tutorial Summary
13. Facts about CHR
As novice, you have to start at the bottom of the ladder:
As novice, you have to start at the bottom of the ladder:

- Learning how to obtain different colors
As novice, you have to start at the bottom of the ladder:

- Learning how to obtain different colors
- by mixing paint
As novice, you have to start at the bottom of the ladder:

- Learning how to obtain different colors
- by mixing paint
- Made easy by the Paint Mixing Rules (PMR) language
Fewer Paint
The Rules
The Rules

Tom Schrijvers (K.U.Leuven)

Constraint Handling Rules
Under the hood, PMR is of course CHR:

<table>
<thead>
<tr>
<th>PMR</th>
<th>CHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>paints</td>
<td>constraints</td>
</tr>
<tr>
<td>rules</td>
<td>simplification rules (textual)</td>
</tr>
<tr>
<td>mixing bucket</td>
<td>constraint store</td>
</tr>
</tbody>
</table>
Declaring our paint colors, in textual form:

```prolog
:- chr_constraint red. %
```

Tom Schrijvers (K.U.Leuven)
Constraint Handling Rules
Declaring our paint colors, in textual form:

```prolog
:- chr_constraint red. %
:- chr_constraint blue. %
```
Declaring our paint colors, in textual form:

:- chr_constraint red. %
:- chr_constraint blue. %
:- chr_constraint yellow. %
:- chr_constraint orange. %

...
Are paint colors really “constraints”? 
Are paint colors really “constraints”? 

**No:** not in the Constraint Programming sense
Are paint colors really “constraints”?  

**No:** not in the Constraint Programming sense  

**Yes:** in the CHR sense  

- 1st-class entities in CHR  
- rewritten by rules
Are paint colors really “constraints”? 

No: not in the Constraint Programming sense

Yes: in the CHR sense
  - 1st-class entities in CHR
  - rewritten by rules

(Are Prolog predicates really “predicates”?)
Simplification Rule: $head \iff body$.
Simplification Rule: $head \iff body$.

“$head$ may be replaced with $body$”
Simplification Rule: $head \iff body$.

"$head$ may be replaced with $body$"

\% red, blue $\iff$ purple.

\% red, yellow $\iff$ orange.

\% blue, yellow $\iff$ green.
Simplification Rule: \( \text{head} \leftrightarrow \text{body} \).

“\text{head} may be replaced with \text{body}”

\[
\begin{align*}
\% & \quad \text{red, blue} \quad \leftrightarrow \quad \text{purple}. \\
\% & \quad \text{red, yellow} \quad \leftrightarrow \quad \text{orange}.
\end{align*}
\]
Simplification Rule: \( \text{head} \leftrightarrow \text{body} \).

“\text{head} may be replaced with \text{body}”

\[
\begin{align*}
\% & \quad \text{red, blue} & \leftrightarrow & \text{purple.} \\
\% & \quad \text{red, yellow} & \leftrightarrow & \text{orange.} \\
\% & \quad \text{blue, yellow} & \leftrightarrow & \text{green.}
\end{align*}
\]
Time For... a cocktail break!
You can have as many heads as you like:

vodka, martini $\leftrightarrow$ vodka_martini.
You can have as many heads as you like:

vodka, martini $\leftrightarrow$ vodka_martini.

tequila, cointreau, lime $\leftrightarrow$ margarita.
You can have as many heads as you like:

vodka, martini \(\leftrightarrow\) vodka\_martini.

tequila, cointreau, lime \(\leftrightarrow\) margarita.

gin, cherry\_brandy, cointreau, grenadine, pineapple, lemon, angostura\_bitters
\(\leftrightarrow\) singapore\_sling.

but at least one!
Warning: too many heads may cause a headache!
Warning: too many heads may cause a headache!

vodka_martini, margarita, singapore_sling
<=> hangover, blackout.
Warning: too many heads may cause a headache!

vodka_martini, margarita, singapore_sling

\[\text{\iff} \text{hangover, blackout.}\]

You can have many constraints in the body too.
Running the Rules

Paint-Mixing Engine:

- Prolog (SWI-Prolog, . . . )
- Leuven CHR system

Everything in a file: `paint.pl`

```
:- use_module(library(chr)).
:- chr_constraint red.
...
red, blue <=> purple.
...
?- [paint].
```

Tom Schrijvers (K.U.Leuven)
Running the Rules

Paint-Mixing Engine:
- Prolog (SWI-Prolog, ...)
- Leuven CHR system

Everything in a file:

paint.pl

```prolog
:- use_module(library(chr)).

:- chr_constraint red.
...

red, blue <=> purple.
...
```
Running the Rules

Paint-Mixing Engine:

- Prolog (SWI-Prolog, ...)
- Leuven CHR system

Everything in a file:

```prolog
paint.pl
:- use_module(library(chr)).
:- chr_constraint red.
...
red, blue <=> purple.
...
?- [paint].
```
Running the Rules

red, blue <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.

?- red, blue.

- queries
- answers
Running the Rules

red, blue <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.

queries

?- red, blue.
purple

?- red, yellow.

answers
Running the Rules

red, blue  <=>  purple.
red, yellow <=> orange.
blue, yellow <=> green.

queries

answers

?- red, blue.
  purple

?- red, yellow.
  orange

?- blue, yellow.
Running the Rules

red, blue  <=>  purple.
red, yellow <=> orange.
blue, yellow <=> green.

- queries
- answers

?- red, blue.
purple

?- red, yellow.
orange

?- blue, yellow.
green
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?- , .

query: left-to-right
red, blue  \iff  purple.
red, yellow  \iff  orange.
blue, yellow  \iff  green.

?- [blue, orange, yellow].

query: left-to-right
active constraint: blue
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?- , .

- query: left-to-right
- active constraint: blue
- put in constraint store
In Slow Motion

red, blue \(\leftrightarrow\) purple.
red, yellow \(\leftrightarrow\) orange.
blue, yellow \(\leftrightarrow\) green.

?- , .

- active constraint: blue
- fire first rule?
- no partner red in constraint store
red, blue $\leftrightarrow$ purple.
red, yellow $\leftrightarrow$ orange.
blue, yellow $\leftrightarrow$ green.

?-

- active constraint: blue
- fire second rule?
- no blue in head
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?-

- active constraint: blue
- fire third rule?
- no partner yellow in constraint store
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?-

, .

- deactivate constraint blue
red, blue  \iff  purple.
red, yellow  \iff  orange.
blue, yellow  \iff  green.
red, blue $\leftrightarrow$ purple.
red, yellow $\leftrightarrow$ orange.
blue, yellow $\leftrightarrow$ green.

?-

- active constraint: yellow
- put in constraint store
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?-

- active constraint: yellow
- fire first rule?
- no yellow in head
red, blue \iff purple.
red, yellow \iff orange.
blue, yellow \iff green.

?- , .

- active constraint: yellow
- fire second rule?
- no partner red in constraint store

Tom Schrijvers (K.U.Leuven)
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?- , .

- active constraint: yellow
- fire third rule?
red, blue  \iff  purple.
red, yellow  \iff  orange.
blue, yellow  \iff  green.

?- active constraint: yellow
- fire third rule!
- with partner blue in constraint store
In Slow Motion

red, blue \(\iff\) purple.
red, yellow \(\iff\) orange.
blue, yellow \(\iff\) green.

?-

\begin{itemize}
\item fire third rule!
\item delete the matched head constraints
\end{itemize}
red, blue $\iff$ purple.
red, yellow $\iff$ orange.
blue, yellow $\iff$ green.

?-

- fire third rule!
- add the body to (front of) query
red, blue  <=>  purple.
red, yellow  <=>  orange.
blue, yellow  <=>  green.

?-  ,  .  

active constraint: green
red, blue $\leftrightarrow$ purple.
red, yellow $\leftrightarrow$ orange.
blue, yellow $\leftrightarrow$ green.

?- , .

... to make a long story short:

- ...  
- final constraint store
What Happens?

red, blue  <=>  purple.
red, yellow  <=>  orange.
blue, yellow  <=>  green.

?- red, yellow.
orange
What Happens?

red, blue  \iff\  purple.
red, yellow  \iff\  orange.
blue, yellow  \iff\  green.

?- red, yellow.
\texttt{orange}

?- yellow, red.
\texttt{orange}
What Happens?

red, blue  <=>  purple.
red, yellow  <=>  orange.
blue, yellow  <=>  green.

?- red.
What Happens?

red, blue  <->  purple.
red, yellow  <->  orange.
blue, yellow  <->  green.

?- red.
red
What Happens?

red, blue <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.

?- red, yellow, blue.
What Happens?

red, blue  <=>  purple.
red, yellow  <=>  orange.
blue, yellow  <=>  green.

?- red, yellow, blue.
  orange
  blue
What Happens?

red, blue  \iff\ purple.
red, yellow  \iff\ orange.
blue, yellow  \iff\ green.

?- red, blue, yellow.
What Happens?

red, blue <=> purple.
red, yellow <=> orange.
blue, yellow <=> green.

?- red, blue, yellow.

purple
yellow
Summary

Simplification Rule:

- \( \text{head} \iff \text{body} \).
- replace head with body
Summary

Simplification Rule:

- \( \text{head} \leftrightarrow \text{body} \).
- replace head with body

Queries:

- processed left-to-right
- incrementally
- active constraint looks for partners
3. Simpagation

For the Sake of Arguments
En Guarde
A Perfect Match
Bigger Programs
Propagation
Order in the Rules
Reactivation
CHR vs. Prolog
Tutorial Summary
Facts about CHR
Brown stays brown.

brown, orange $\iff$ brown.
brown, green $\iff$ brown.
brown, purple $\iff$ brown.
...

Simpagation rule: $head_k \setminus head_r \iff body$.

brown \ orange \iff true.
brown \ green \iff true.
brown \ purple \iff true.
...

Simpagination rule: $head_k \setminus head_r \iff body$.

brown \ orange \iff true.
brown \ green \iff true.
brown \ purple \iff true.
...

- $head_k$: kept head (1 or more)
Simpagation rule: $head_k \setminus head_r \iff body$.

- brown \ orange $\iff$ true.
- brown \ green $\iff$ true.
- brown \ purple $\iff$ true.

- $head_k$ : kept head (1 or more)
- $head_r$ : removed head (1 or more)
Simpagation rule: \( \text{head}_k \setminus \text{head}_r \iff \text{body} \). 

\[
\begin{align*}
\text{brown} \setminus \text{orange} & \iff \text{true}. \\
\text{brown} \setminus \text{green} & \iff \text{true}. \\
\text{brown} \setminus \text{purple} & \iff \text{true}. \\
\ldots
\end{align*}
\]

- \( \text{head}_k \): kept head (1 or more)
- \( \text{head}_r \): removed head (1 or more)
- \( \text{true} \): Prolog’s no-op
philosophers\_stone \(\setminus\) lead

\(\iff\) gold.

?- philosophers\_stone, lead.
philosophers_stone \ lead
  <=> gold.

?- philosophers_stone, lead.
philosophers_stone
  gold
philosophers_stone \ lead
  <=> gold.

?- philosophers_stone, lead.
philosophers_stone
gold

?- philosophers_stone, lead, lead.
philosophers_stone \ lead
     <=>  gold.

?- philosophers_stone, lead.
philosophers_stone
gold

?- philosophers_stone, lead, lead.
philosophers_stone
gold
gold
philosophers_stone \ lead \n  \n  \n<=> gold.

?- lead, lead.
philosophers_stone \ lead
    <=> gold.

?- lead, lead.
lead
lead
philosophers\_stone \ lead
  \ <=> \ gold.

?- lead, lead.
lead
lead

?- lead, lead, philosophers\_stone.
philosophers_stone \ lead  

<=>  gold.

?- lead, lead.
lead
lead

?- lead, lead, philosophers_stone.
philosophers_stone

 gold
gold
philosophers_stone \ lead
  \=\> gold.

?- lead, lead.
lead
lead

?- lead, lead, philosophers_stone.
philosophers_stone
gold
gold

Rule fires with all combinations!
Summary

Simplification Rule:
- head $\iff$ body.
- replace head with body

Tom Schrijvers (K.U.Leuven)
Simplification Rule:

- $head \iff body$.
- replace head with body

Simpagation Rule:

- $head_k \setminus head_r \iff body$.
- replace $head_r$ with $body$
- in the presence of $head_k$
- fires all possible combinations
<table>
<thead>
<tr>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

- constraints can have arguments
Piggy Bank Merger

:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

- constraints can have arguments
- arity must be declared
Piggy Bank Merger

:- chr_constraint piggy/1.
piggy(I), piggy(J) <=> K is I + J, piggy(K).

- constraints can have arguments
- arity must be declared
- Prolog terms as arguments
Piggy Bank Merger

:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

?- piggy(5), piggy(1), piggy(4), piggy(2).
:- chr_constraint piggy/1.
piggy(I), piggy(J) <=> K is I + J, piggy(K).
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

?- , , , .
:- chr_constraint piggy/1.
piggy(I), piggy(J) <=> K is I + J, piggy(K).
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

?- 4€, 2€, 6€.
Piggy Bank Merger

:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

?- , , , , .
:- chr_constraint piggy/1.

piggy(I), piggy(J) <= K is I + J, piggy(K).
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

?-  
  , , , , .
:- chr_constraint piggy/1.

piggy(I), piggy(J) <=> K is I + J, piggy(K).

?- , , , , .

12€

Tom Schrijvers (K.U.Leuven)
CHR is an embedded language

- embedded in host language $\mathcal{X}$
CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X} = \text{Prolog}$
CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X} = \text{Prolog}$

2-way communication:
CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X} = \text{Prolog}$

2-way communication:

- Prolog calls CHR constraints
e.g. all along from toplevel
CHR is an embedded language

- embedded in host language $\mathcal{X}$
- here $\mathcal{X} = \text{Prolog}$

2-way communication:

- Prolog calls CHR constraints
  - e.g. all along from toplevel
- CHR calls Prolog
  - e.g. $K$ is $I + J$
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).

?- value([a]), value([b]).
Not Just Numbers

```prolog
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).

?- value([a]), value([b]).
value([b,a]).
```
Not Just Numbers

:- chr_constraint value/1.

value(I), value(J) <= append(I,J,K), value(K).

?- value([a]), value([b]).
value([b,a]).

?- value([f(a),g(b)]), value([h(i,j)]).
Not Just Numbers

:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).

?- value([a]), value([b]).
value([b,a]).

?- value([f(a),g(b)]), value([h(i,j)]).
value([h(i,j),f(a),g(b)]).
:- chr_constraint value/1.

value(I), value(J) <=> append(I,J,K), value(K).

?- value([a]), value([b]).
value([b,a]).

?- value([f(a),g(b)]), value([h(i,j)]).
value([h(i,j),f(a),g(b)]).

Any Prolog terms!
CHR constraints

- zero or more arguments
- Prolog terms
Summary

CHR constraints
- zero or more arguments
- Prolog terms

Bodies
- CHR constraints and
- Prolog predicate calls
Overview

1. Introduction
2. The Art of CHR
3. Simpagation
4. For the Sake of Arguments
5. En Guarde
6. A Perfect Match
7. Bigger Programs
8. Propagation
9. Order in the Rules
10. Reactivation
11. CHR vs. Prolog
12. Tutorial Summary
13. Facts about CHR
Count Down

Problem: write a program to enumerate values:

?- generate(5).
Problem: write a program to enumerate values:

?- generate(5).

value(5)
value(4)
value(3)
value(2)
value(1)
Solution:

:- chr_constraint generate/1.
Solution:

:- chr_constraint generate/1.
:- chr_constraint value/1.
Solution:

:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=>
    value(N), M is N - 1, generate(M).
Running the Program

```
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <= value(N), M is N - 1, generate(M).

?- generate(5).
```

% waiting
% still waiting

ERROR: Out of local stack
Recursion without a base case!
Running the Program

```prolog
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).

?- generate(5).
   % waiting
```

ERROR: Out of local stack
Recursion without a base case!
Running the Program

```prolog
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).

?- generate(5).
   % waiting
   % still waiting
```

ERROR: Out of local stack
Recursion without a base case!
Running the Program

:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).

?- generate(5).

% waiting
% still waiting
ERROR: Out of local stack
Running the Program

```prolog
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> value(N), M is N - 1, generate(M).

?- generate(5).
    % waiting
    % still waiting
ERROR: Out of local stack
```

Recursion without a base case!
Guarded Rules

:- chr_constraint generate/1.
:- chr_constraint value/1.

\[
\begin{align*}
\text{generate}(N) & \iff N = 0 \mid \text{true}. \\
\text{generate}(N) & \iff N > 0 \mid \text{value}(N), \\
& \quad \text{M is } N - 1, \text{ generate}(M).
\end{align*}
\]
Guarded Rules

:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0 | value(N),
    M is N - 1, generate(M).

Guarded rule:

- apply rule if guard succeeds
Guarded Rules

:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0 | value(N),
      M is N - 1, generate(M).

Guarded rule:

- apply rule if guard succeeds
- guard is optional
Guarded Rules

:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(N) <=> N == 0 | true.
generate(N) <=> N > 0 | value(N),
               M is N - 1, generate(M).

Guarded rule:

- apply rule if guard succeeds
- guard is optional
- guard may contain any Prolog, but may not bind any variables from the head (pure check)
generate(N) <=> N > 0 | value(N),
    M is N - 1, generate(M).
generate(N) <=> N > 0 | value(N),
        M is N - 1, generate(M).

?- generate(5).
generate(N) <=> N > 0 | value(N),
                 M is N - 1, generate(M).

?- generate(5).
value(5)
value(4)
value(3)
value(2)
value(1)
generate(0)
Simplification Rule:

- $head \iff guard \mid body$.
- replace head with body
- if $guard$ succeeds
**Simplification Rule:**

- $head \leftrightarrow guard \mid body$.
- replace head with body
- if $guard$ succeeds

**Simpagation Rule:**

- $head_k \backslash head_r \leftrightarrow guard \mid body$.
- replace $head_r$ with $body$
- in the presence of $head_k$
- if $guard$ succeeds
1 Introduction
2 The Art of CHR
3 Simpagation
4 For the Sake of Arguments
5 En Guarde
6 A Perfect Match
7 Bigger Programs
8 Propagation
9 Order in the Rules
10 Reactivation
11 CHR vs. Prolog
12 Tutorial Summary
13 Facts about CHR
:- chr_constraint generate/1.
:- chr_constraint value/1.

generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
              M is N - 1, generate(M).

Matching:

- “inline” notation for guard
- same meaning as explicit guard
<table>
<thead>
<tr>
<th>Matching</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(X) &lt;=&gt; true.</td>
<td>c(X) &lt;=&gt; true</td>
</tr>
</tbody>
</table>
### Meaning of Matching

<table>
<thead>
<tr>
<th>Maching</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(X) (\iff) true.</td>
<td>c(X) (\iff) true</td>
</tr>
<tr>
<td>c(a) (\iff) true.</td>
<td>c(X) (\iff) (X == a)</td>
</tr>
</tbody>
</table>

Tom Schrijvers (K.U.Leuven)
### Meaning of Matching

<table>
<thead>
<tr>
<th>Matching</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c(X) &lt;=&gt; true.</code></td>
<td>`c(X) &lt;=&gt; true</td>
</tr>
<tr>
<td><code>c(a) &lt;=&gt; true.</code></td>
<td>`c(X) &lt;=&gt; X == a</td>
</tr>
<tr>
<td><code>c(f(A)) &lt;=&gt; true.</code></td>
<td>`c(X) &lt;=&gt; nonvar(X), X = f(A)</td>
</tr>
</tbody>
</table>
## Meaning of Matching

<table>
<thead>
<tr>
<th>Matching</th>
<th>Guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(X) (\iff) true.</td>
<td>c(X) (\iff) true \mid true.</td>
</tr>
<tr>
<td>c(a) (\iff) true.</td>
<td>c(X) (\iff) (X == a) \mid true.</td>
</tr>
<tr>
<td>c(f(A)) (\iff) true.</td>
<td>c(X) (\iff) nonvar(X), (X = f(A)) \mid true.</td>
</tr>
<tr>
<td>c(X,X) (\iff) true.</td>
<td>c(X,Y) (\iff) (X == Y) \mid true.</td>
</tr>
<tr>
<td>Matching</td>
<td>Guard</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td><code>c(X) &lt;=&gt; true</code></td>
<td>`c(X) &lt;=&gt; true</td>
</tr>
<tr>
<td><code>c(a) &lt;=&gt; true</code></td>
<td>`c(X) &lt;=&gt; X == a</td>
</tr>
<tr>
<td><code>c(f(A)) &lt;=&gt; true</code></td>
<td>`c(X) &lt;=&gt; nonvar(X), X = f(A)</td>
</tr>
<tr>
<td><code>c(X,X) &lt;=&gt; true</code></td>
<td>`c(X,Y) &lt;=&gt; X == Y</td>
</tr>
<tr>
<td><code>c(X), d(X) &lt;=&gt; true</code></td>
<td>`c(X), d(Y) &lt;=&gt; X == Y</td>
</tr>
</tbody>
</table>
Matching: CHR vs. Prolog

\begin{verbatim}
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
\end{verbatim}
Matching: CHR vs. Prolog

\[ p(\text{world}) \leftarrow \text{writeln('Hello, World!')} \].

\[ :- \text{chr\_constraint c/1}. \]

\[ c(\text{world}) \iff \text{writeln('Hello, World!')} \].

?- p(\text{world}).
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.

c(world) <=> writeln('Hello, World!').

?- p(world).
Hello, World!
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').

?- p(world).
Hello, World!

?- p(Free).

?- c(world).
Hello, World!

?- c(Free).

?- c(hello).
No
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').

?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world
Matching: CHR vs. Prolog

\[
p(\text{world}) :- \text{writeln('Hello, World!')}.\\[10pt]
\text{:- chr_constraint c/1.}\\[10pt]c(\text{world}) \iff \text{writeln('Hello, World!')}.\\[10pt]\\[10pt]\]

?- p(\text{world}).
Hello, World!

?- p(\text{Free}).
Hello, World!
\text{Free} = \text{world}

?- p(\text{hello}).
Matching: CHR vs. Prolog

```
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

```
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No
```
Matching: CHR vs. Prolog

\[ p(\text{world}) :- \text{writeln('Hello, World!')} \]. \]
\[ :- \text{chr_constraint c/1}. \]
\[ c(\text{world}) \iff \text{writeln('Hello, World!')} \]. \]

?- \ p(\text{world}).
Hello, World!

?- \ p(\text{Free}).
Hello, World!
\text{Free} = \text{world}

?- \ p(\text{hello}).
\text{No}
Matching: CHR vs. Prolog

```prolog
p(world) :- writeln('Hello, World!').
:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
```

?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No

?- c(world).
Hello, World!
Matching: CHR vs. Prolog

\( p(\text{world}) :- \text{writeln('Hello, World!')} \).

\( :- \text{chr_constraint c/1.} \)

\( c(\text{world}) \iff \text{writeln('Hello, World!')} \).

\( ?- p(\text{world}). \)
\( \text{Hello, World!} \)

\( ?- p(\text{Free}). \)
\( \text{Hello, World!} \)
\( \text{Free = world} \)

\( ?- p(\text{hello}). \)
\( \text{No} \)

\( ?- c(\text{world}). \)
\( \text{Hello, World!} \)

\( ?- c(\text{Free}). \)
p(world) :- writeln('Hello, World!').

:- chr_constraint c/1.
c(world) <-> writeln('Hello, World!').

?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No

?- c(world).
Hello, World!

?- c(Free).
c(Free)
Matching: CHR vs. Prolog

\begin{verbatim}
p(world) :- writeln('Hello, World!').
:- chr_constraint c/1.
c(world) <=> writeln('Hello, World!').
\end{verbatim}

\begin{verbatim}
?- p(world).
Hello, World!

?- p(Free).
Hello, World!
Free = world

?- p(hello).
No

?- c(world).
Hello, World!

?- c(Free).
c(Free)

?- c(hello).
\end{verbatim}
Matching: CHR vs. Prolog

\[ p(\text{world}) :- \text{writeln}(\text{'Hello, World!'}). \]

\[ :- \text{chr\_constraint} \ c/1. \]
\[ c(\text{world}) \iff \text{writeln}(\text{'Hello, World!'}). \]

?- p(\text{world}).
Hello, World!

?- p(\text{Free}).
Hello, World!
Free = world

?- p(\text{hello}).
No

?- c(\text{world}).
Hello, World!

?- c(\text{Free}).
c(\text{Free})

?- c(\text{hello}).
c(\text{hello})
Summary

Matching
- short-hand for equality-based guards
- one-way unification
  - only instantiates head
  - does not instantiate constraints
1. Introduction
2. The Art of CHR
3. Simpagation
4. For the Sake of Arguments
5. En Guarde
6. A Perfect Match
7. Bigger Programs
8. Propagation
9. Order in the Rules
10. Reactivation
11. CHR vs. Prolog
12. Tutorial Summary
13. Facts about CHR
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

What’s the outcome?
?- generate(4).
Combining Rules

generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                 M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

What’s the outcome?

?- generate(4).
value(10)
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
              M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

What’s the outcome?

?- generate(4).
value(10)

?- generate(5).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
              M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

What’s the outcome?

?- generate(4).
value(10)

?- generate(5).
value(15)
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                 M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
    M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
   M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

?- M is 2 - 1
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
               M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
              M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                  M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                        M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

?- M is 1 - 1
   2 1
   M
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
               M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
In Slow Motion

generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
               M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).

?-
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
           M is N - 1, generate(M).

value(I), value(J) <=> K is I + J, value(K).
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
               M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.

What’s the outcome?
?- generate(4).
Combining Rules

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                  M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.

What’s the outcome?

?- generate(4).
value(2)
value(3)
Combining Rules

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                 M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.

What’s the outcome?

?- generate(4).
value(2)
value(3)

?- generate(10).
Combining Rules

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),

\[ M \text{ is } N - 1, \text{ generate}(M). \]

value(I) \ value(J) <=> J mod I =:= 0 | true.

What’s the outcome?

?- generate(4).
value(2)
value(3)

?- generate(10).
value(2)
value(3)
value(5)
value(7)
How Does It Work?

```
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
    M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```
How Does It Work?

\[
\begin{align*}
generate(1) & \iff \text{true}. \\
generate(N) & \iff N > 1 \mid value(N), \\
& \quad M \text{ is } N - 1, generate(M). \\
value(I) \setminus value(J) & \iff J \mod I = 0 \mid \text{true}. 
\end{align*}
\]
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                    M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

generate(1) \<=>\ true.
generate(N) \<=>\ N > 1 | value(N),
        M is N - 1, generate(M).

value(I) \ { value(J) \<=>\ J mod I = := 0 | true. \}
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1  |  value(N),
                 M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I :::= 0 | true.
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
               M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
    M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

```plaintext
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
    M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Tom Schrijvers (K.U.Leuven)
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                 M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
               M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
               M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

\[
\text{generate}(1) \iff \text{true}.
\]
\[
\text{generate}(N) \iff N > 1 \mid \text{value}(N),
\]
\[
M \text{ is } N - 1, \text{generate}(M).
\]
\[
\text{value}(I) \backslash \text{value}(J) \iff J \mod I =:= 0 \mid \text{true}.
\]
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
               M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
How Does It Work?

generate(1) \iff true.
generate(N) \iff N > 1 \land value(N),
M = N - 1, generate(M).

value(I) \setminus value(J) \iff J \mod I = 0 \lor true.
How Does It Work?

generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                 M is N - 1, generate(M).

value(I) \
value(J) <=> J mod I =:= 0 | true.
Alternatives?

\[\text{generate}(1) \iff \text{true.} \]
\[\text{generate}(N) \iff N > 1 \mid \text{value}(N), \quad \text{M is } N - 1, \text{ generate}(M). \]

\[\text{value}(I) \setminus \text{value}(J) \iff J \mod I =: 0 \mid \text{true.}\]

Just added value(3).
What now?
Alternatives?

```prolog
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                 M is N - 1, generate(M).
```

```prolog
value(I) \ value(J) <=> J mod I =:= 0 | true.
```

Just added value(3).
What now?

- remove value(6)?
Alternatives?

\[
\text{generate}(1) \iff \text{true}.
\]
\[
\text{generate}(N) \iff N > 1 \quad \text{|} \quad \text{value}(N),
\]
\[
\text{M is } N - 1, \text{ generate}(M).
\]

\[
\text{value}(I) \setminus \text{value}(J) \iff J \mod I = 0 \quad \text{|} \quad \text{true}.
\]

Just added value(3).
What now?

- remove value(6) ?
- remove value(9) ?
Alternatives?

\[
\begin{align*}
generate(1) & \iff \text{true}. \\
generate(N) & \iff N > 1 \mid value(N), \\
 & \quad M \text{ is } N - 1, \ generate(M). \\
value(I) \ \checkmark \ value(J) & \iff J \mod I = 0 \mid \text{true}. 
\end{align*}
\]

Just added \text{value}(3). What now?

- remove \text{value}(6)?
- remove \text{value}(9)?
- unspecified in CHR!
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I :== 0 | true.
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
    M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
            M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                  M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I =:= 0 | true.
generate(1) <=> true.
generate(N) <=> N > 1 | value(N),
                  M is N - 1, generate(M).

value(I) \ value(J) <=> J mod I == 0 | true.
CHR Programs

- Programs consist of a sequence of rules
CHR Programs

- Programs consist of a sequence of rules
- Active constraint traverses the rules top-to-bottom to find any that fire
CHR Programs

- Programs consist of a sequence of rules
- Active constraint traverses the rules top-to-bottom to find any that fire
- Rest of query/body waits
CHR Programs

- Programs consist of a sequence of rules
- Active constraint traverses the rules top-to-bottom to find any that fire
- Rest of query/body waits
- Unspecified which partner is found
Simpagation with $H_k = \emptyset$

Simpagation rule

$\text{head}_k \setminus \text{head}_r \iff \text{guard} \mid \text{body}$.
Simpagation with $H_k = \emptyset$

**Simpagation rule**

$head_k \setminus head_r \iff guard \mid body$.

What if $head_k = \emptyset$?
Simpagation with $H_k = \emptyset$

**Simpagation rule**

$$\text{head}_k \setminus \text{head}_r \leftrightarrow \text{guard} \mid \text{body}.$$ 

What if $\text{head}_k = \emptyset$?  

**Simplification rule**

$$\text{head}_r \leftrightarrow \text{guard} \mid \text{body}.$$
Simpagination with $H_r = \emptyset$

**Simpagination rule**

$$\text{head}_k \ \backslash \ \text{head}_r \iff \text{guard} \ | \ \text{body}.$$
Simpagation with $H_r = \emptyset$

**Simpagation rule**

$$\text{head}_k \, \backslash \, \text{head}_r \iff \text{guard} \mid \text{body}.$$  

What if $\text{head}_r = \emptyset$?
Simpagation with $H_r = \emptyset$

**Simpagation rule**

$$head_k \backslash head_r \iff guard \mid body.$$  

What if $head_r = \emptyset$?

**Propagation rule**

$$head_k \Rightarrow guard \mid body.$$  

- add body
- in the presence of $head_k$
- if $guard$ holds
Propagation

\[ a, b \Rightarrow c. \]

?- a, b.
a, b ==> c.

?- a, b.

a
b
c
Propagation

\[ \text{a, b} \Rightarrow \text{c.} \]

?- a, b.

\[ \text{a} \]

\[ \text{b} \]

\[ \text{c} \]

?- a, b, b.
Proposition

\[ a, b \Rightarrow c. \]

?- a, b.

?- a, b, b.
Writing Wolfram’s cellular automaton Rule 110 in CHR.
Writing Wolfram’s cellular automaton Rule 110 in CHR.
Writing Wolfram’s cellular automaton Rule 110 in CHR.
Writing Wolfram’s cellular automaton Rule 110 in CHR.
Writing Wolfram’s celluar automaton Rule 110 in CHR.
Cellular Automaton

Writing Wolfram’s celluar automaton Rule 110 in CHR.
Writing Wolfram’s cellular automaton Rule 110 in CHR.
Writing Wolfram’s cellular automaton Rule 110 in CHR.
Cellular Automaton

Writing Wolfram’s cellular automaton Rule 110 in CHR.

Tom Schrijvers (K.U.Leuven)
CHR cells

% black(Name,Below,Above,Generation).
:- chr_constraint black/4.

% white(Name,Below,Above,Generation).
:- chr_constraint white/4.
CHR cells

\% black(\text{Name}, \text{Below}, \text{Above}, \text{Generation}).
:- \text{chr\_constraint} \text{ black}/4.

\% white(\text{Name}, \text{Below}, \text{Above}, \text{Generation}).
:- \text{chr\_constraint} \text{ white}/4.

?- \text{white}(d, c, a, 1),
   \text{black}(c, b, d, 1),
   \text{white}(b, a, c, 1),
   \text{white}(a, d, b, 1).
\text{white}(C,\_\_\_G)\ , \\
\text{black}(B,A,C,G)\ , \\
\text{white}(A,\_\_\_G) \\
\implies G < 10 \mid \text{NG is } G + 1, \text{black}(B,A,C,\text{NG}).
white(C,_,_,G) ,
black(B,A,C,G) ,
white(A,_,_,G)

\[ \Rightarrow G < 10 \mid \text{NG is } G + 1, \text{ black}(B,A,C,\text{NG}). \]

?- white(d,c,a,1),
   black(c,b,d,1),
   white(b,a,c,1),
   white(a,d,b,1).
Automaton Rule

white(C,_,_,G) ,
black(B,A,C,G) ,
white(A,_,_,G)

implies G < 10 | NG is G + 1, black(B,A,C,NG).

?- white(d,c,a,1),
   black(c,b,d,1),
   white(b,a,c,1),
   white(a,d,b,1).
generate(N) ==> value(2).
Another Example of Propagation

generate(N) ==> value(2).

generate(N), value(I) ==>  
    I < N | J is I + 1, value(J).  

Another Example of Propagation

generate(N) ==> value(2).

generate(N), value(I) ==> 
    I < N | J is I + 1, value(J).

value(I) \ value(J) <=> J mod I =:= 0 | true.

?- generate(10).
Another Example of Propagation

generate(N) ==> value(2).

generate(N), value(I) ==> 
    I < N | J is I + 1, value(J).

value(I) \ value(J) <==> J mod I =:= 0 | true.

?- generate(10).
value(2)
value(3)
value(5)
value(7)
generate(10)
Summary

Simplification Rule: \( \text{head} \iff \text{guard} \mid \text{body} \).
- replace head with body

Simpagation Rule: \( \text{head}_k \backslash \text{head}_r \iff \text{guard} \mid \text{body} \).
- replace \( \text{head}_r \) with \( \text{body} \)
- in the presence of \( \text{head}_k \)

Propagation Rule: \( \text{head} \implies \text{guard} \mid \text{body} \).
- add \( \text{body} \)
- in the presence of \( \text{head} \)
- fires once for each combination (propagation history)
Biased Coin?

coin <=> heads.
coin <=> tails.

What’s the outcome?

?- coin.
Biased Coin?

```prolog
coin <=> heads.
coin <=> tails.
```

What's the outcome?

```prolog
?- coin.
heads
```
Biased Coin?

\[
\begin{align*}
\text{coin} & \leftrightarrow \text{heads}. \\
\text{coin} & \leftrightarrow \text{tails}.
\end{align*}
\]

What’s the outcome?

?- coin.

\textbf{heads}

Rules are tried in order!
Coin Flipping in Prolog

<table>
<thead>
<tr>
<th>CHR</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>coin &lt;=&gt; heads.</td>
<td>p_coin :- heads.</td>
</tr>
<tr>
<td>coin &lt;=&gt; tails.</td>
<td>p_coin :- tails.</td>
</tr>
</tbody>
</table>

What’s the outcome?
Coin Flipping in Prolog

CHR

<table>
<thead>
<tr>
<th>Rule</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>coin &lt;= heads</td>
<td>p_coin :- heads.</td>
</tr>
<tr>
<td>coin &lt;= tails</td>
<td>p_coin :- tails.</td>
</tr>
</tbody>
</table>

What’s the outcome?

?- coin.
  heads
Coin Flipping in Prolog

CHR

\[
\begin{align*}
\text{coin} & \leftrightarrow \text{heads}. \\
\text{coin} & \leftrightarrow \text{tails}.
\end{align*}
\]

Prolog

\[
\begin{align*}
\text{p} \_\text{coin} & : - \text{heads}. \\
\text{p} \_\text{coin} & : - \text{tails}.
\end{align*}
\]

What’s the outcome?

?- \text{coin}.
\text{heads}

?- \text{p} \_\text{coin}.
\text{heads}
### Coin Flipping in Prolog

**CHR**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coin &lt;=&gt; heads.</td>
<td>p_coin :- heads.</td>
</tr>
<tr>
<td>coin &lt;=&gt; tails.</td>
<td>p_coin :- tails.</td>
</tr>
</tbody>
</table>

**What’s the outcome?**

<table>
<thead>
<tr>
<th>Prolog Query</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>?- coin.</td>
<td>heads</td>
</tr>
<tr>
<td>?- p_coin.</td>
<td>heads ; y</td>
</tr>
</tbody>
</table>

Prolog: backtracking

CHR: committed choice for simplification rules
Coin Flipping in Prolog

CHR

\[
\begin{align*}
\text{coin} & \iff \text{heads}. \\
\text{coin} & \iff \text{tails}.
\end{align*}
\]

Prolog

\[
\begin{align*}
p\_\text{coin} & : = \text{heads}. \\
p\_\text{coin} & : = \text{tails}.
\end{align*}
\]

What’s the outcome?

?- \text{coin}.
\begin{verbatim}
heads
\end{verbatim}

?- p\_coin.
\begin{verbatim}
heads ; y
\end{verbatim}
\begin{verbatim}
tails
\end{verbatim}
### Coin Flipping in Prolog

<table>
<thead>
<tr>
<th>CHR</th>
<th>Prolog</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{coin} \iff \text{heads}.)</td>
<td>(\text{p_coin} \leftarrow \text{heads}.)</td>
</tr>
<tr>
<td>(\text{coin} \iff \text{tails}.)</td>
<td>(\text{p_coin} \leftarrow \text{tails}.)</td>
</tr>
</tbody>
</table>

#### What’s the outcome?

```prolog
?- coin.
heads

?- p_coin.
heads ; y
tails
```

- **Prolog:** backtracking
- **CHR:** committed choice for simplification rules
Propagation Coin Flipping

\begin{verbatim}
coin ==> heads.
coin ==> tails.
\end{verbatim}

What’s the outcome?

?- coin.
heads
tails
coin
propagation coin flipping

coin ==> heads.
coin ==> tails.

What's the outcome?

?- coin.
heads
tails
coin

Rules are applied in sequence.
Propagation Coin Flipping

\( \text{coin} \Rightarrow \text{heads.} \)
\( \text{coin} \Leftrightarrow \text{tails.} \)
\( \text{coin} \Rightarrow \text{side.} \)

What's the outcome?

?- coin.
heads
tails
Propagation Coin Flipping

\[
\text{coin} \Rightarrow \text{heads.} \\
\text{coin} \Leftrightarrow \text{tails.} \\
\text{coin} \Rightarrow \text{side.}
\]

**What’s the outcome?**

?- coin.

heads
tails

**Rules are applied in sequence, until the active constraint is removed.**
generate(0) $\leftrightarrow$ true.
generate(N) $\leftrightarrow$ N > 0 | value(N),
                      M is N - 1, generate(M).
Exploiting Rule Order

```prolog
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                 M is N - 1, generate(M).
```

vs.

```prolog
generate(0) <=> true.
generate(N) <=> value(N),
                 M is N - 1, generate(M).
```
Exploiting Rule Order

generate(0) <= true.
generate(N) <=> N > 0 | value(N),
                M is N - 1, generate(M).

vs.

generate(0) <= true.
generate(N) <=> value(N),
                M is N - 1, generate(M).

for queries ?- generate(N). with N ≥ 0
Exploiting Rule Order

```
generate(0) <=> true.
generate(N) <=> N > 0 | value(N),
                   M is N - 1, generate(M).
```

vs.

```
generate(0) <=> true.
generate(N) <=> value(N),
                   M is N - 1, generate(M).
```

for queries `- generate(N).` with \(N \geq 0\)

Better Style?
Exploiting Rule Order

\[
\begin{align*}
generate(0) & \iff \text{true}. \\
generate(N) & \iff N > 0 \mid \text{value}(N), \\
& \hspace{1cm} M \text{ is } N - 1, \text{generate}(M). \\
\end{align*}
\]

vs.

\[
\begin{align*}
generate(0) & \iff \text{true}. \\
generate(N) & \iff \text{value}(N), \\
& \hspace{1cm} M \text{ is } N - 1, \text{generate}(M). \\
\end{align*}
\]

for queries \(-\) \text{generate}(N). \text{ with } N \geq 0

Better Style?

Compare to:

\[
\begin{align*}
generate(0) & :- !. \\
generate(N) & :- \text{value}(N), M \text{ is } N - 1, \text{generate}(M). \\
\end{align*}
\]
Rule Order

- rules are tried from top to bottom
- applied in sequence
- until the active constraint is removed
- committed choice: no alternatives explored for simplification rules
1. Introduction
2. The Art of CHR
3. Simpagation
4. For the Sake of Arguments
5. En Guarde
6. A Perfect Match
7. Bigger Programs
8. Propagation
9. Order in the Rules
10. Reactivation
11. CHR vs. Prolog
12. Tutorial Summary
13. Facts about CHR
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).

?- c(hello), c(world).
Hello, World!

c(hello) <=> writeln(hello).
c(world) <=> writeln(world).

?- c(hello), c(world).
hello
world
Hello, World!

\[ \text{c(} \text{hello} \text{)} \iff \text{writeln(} \text{hello} \text{)}. \]
\[ \text{c(} \text{world} \text{)} \iff \text{writeln(} \text{world} \text{)}. \]

```prolog
?- c(\text{hello}), c(\text{world}).
\text{hello}
\text{world}
?- c(X).
```

Tom Schrijvers (K.U.Leuven)
Hello, World!

c(\text{hello}) \iff \text{writeln(hello)}.
c(\text{world}) \iff \text{writeln(world)}.

?- c(\text{hello}), c(\text{world}).
\text{hello}
\text{world}

?- c(X).
c(X)
Hello, World!

c(hello) <=> writeln(hello).
c(world) <=> writeln(world).

?- c(hello), c(world).
hello
world
?- c(X).
c(X)
?- c(X), X = hello.
Hello, World!

c(hello) <=> writeln(hello).
c(world) <=> writeln(world).

?- c(hello), c(world).
   hello
   world
?- c(X).
   c(X)
?- c(X), X = hello.
   hello
Hello, World!

c(hello) <=> writeln(hello).
c(world) <=> writeln(world).

?- c(hello), c(world).
  hello
  world
?- c(X).
  c(X)
?- c(X), X = hello.
  hello
?- c(X), c(world), X = hello.
c(hello) <=> writeln(hello).
c(world) <=> writeln(world).

?- c(hello), c(world).
hello
world
?- c(X).
c(X)
?- c(X), X = hello.
hello
?- c(X), c(world), X = hello.
world
hello
X = hello

Reactivation
▶ constraints suspend in the constraint store
▶ unification reactivates suspended constraints
A Constraint: Inequality

neq(X,X) <=> fail.
neq(X,Y) <=> X \neq Y | true.

?- neq(a,a).

?- neq(a,b).

?- neq(A,B).

?- neq(A,B), A = B.

?- neq(A,B), A = a, B = a.

?- neq(A,B), A = a, B = b.

?- neq(A,B), A = f(C), B = f(D).

neq(f(C),f(D))
A Constraint: Inequality

\[
\begin{align*}
\text{neq}(X,X) & \iff \text{fail}. \\
\text{neq}(X,Y) & \iff X \neq Y \mid \text{true}.
\end{align*}
\]

?- \text{neq}(a,a).
No
A Constraint: Inequality

\texttt{neq(X,X) \iff \text{fail.}}
\texttt{neq(X,Y) \iff X \neq Y \mid \text{true.}}

?- \texttt{neq(a,a).}
\textbf{No}

?- \texttt{neq(a,b).}
A Constraint: Inequality

\[ \text{neq}(X,X) \leftrightarrow \text{fail.} \]
\[ \text{neq}(X,Y) \leftrightarrow X \neq Y | \text{true.} \]

?- neq(a,a).
No

?- neq(a,b).
Yes
A Constraint: Inequality

\[ neq(X,X) \iff \text{fail.} \]
\[ neq(X,Y) \iff X \neq Y \mid \text{true.} \]

?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(f(C),f(D))
A Constraint: Inequality

\[ \text{neq}(X,X) \iff \text{fail.} \]
\[ \text{neq}(X,Y) \iff X \neq Y \mid \text{true.} \]

?- \text{neq}(A,B), A = B.

?- \text{neq}(a,a).
No

?- \text{neq}(a,b).
Yes

?- \text{neq}(A,B).
\text{neq}(A,B) \]
A Constraint: Inequality

\[
\text{neq}(X,X) \iff \text{fail.}
\]
\[
\text{neq}(X,Y) \iff X \neq Y \mid \text{true.}
\]

?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
\text{neq}(A,B)
A Constraint: Inequality

neq(X,X) <=> fail.
neq(X,Y) <=> X \neq Y | true.

?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)

?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
A Constraint: Inequality

\[ \text{neq}(X, X) \iff \text{fail}. \]
\[ \text{neq}(X, Y) \iff X \neq Y \lor \text{true}. \]

?- neq(a,a).
No

?- neq(a,b).
Yes

?- neq(A,B).
neq(A,B)

?- neq(A,B), A = B.
No

?- neq(A,B), A = a, B = a.
No
A Constraint: Inequality

\[
\text{neq}(X,X) \iff \text{fail}.
\]
\[
\text{neq}(X,Y) \iff X \neq Y \mid \text{true}.
\]

?- \text{neq}(a,a).
No

?- \text{neq}(a,b).
Yes

?- \text{neq}(A,B).
\text{neq}(A,B)
A Constraint: Inequality

\[
\text{neq}(X,X) \iff \text{fail}.
\]
\[
\text{neq}(X,Y) \iff X \neq Y \land \text{true}.
\]

?- \text{neq}(a,a).
No

?- \text{neq}(a,b).
Yes

?- \text{neq}(A,B).
\text{neq}(A,B)

?- \text{neq}(A,B), A = B.
No

?- \text{neq}(A,B), A = a, B = a.
No

?- \text{neq}(A,B), A = a, B = b.
Yes
A Constraint: Inequality

\[\text{neq}(X, X) \iff \text{fail}.\]
\[\text{neq}(X, Y) \iff X \neq Y \mid \text{true}.\]

?- \text{neq}(a, a).
No

?- \text{neq}(a, b).
Yes

?- \text{neq}(A, B).
\text{neq}(A, B)

?- \text{neq}(A, B), A = B.
No

?- \text{neq}(A, B), A = a, B = a.
No

?- \text{neq}(A, B), A = a, B = b.
Yes

?- \text{neq}(A, B), A = f(C), B = f(D).
A Constraint: Inequality

\[ \text{neq}(X,X) \iff \text{fail.} \]
\[ \text{neq}(X,Y) \iff X \neq Y \mid \text{true.} \]

?- neq(a,a).
No
?- neq(a,b).
Yes
?- neq(A,B).
neq(A,B)

?- neq(A,B), A = B.
No
?- neq(A,B), A = a, B = a.
No
?- neq(A,B), A = a, B = b.
Yes
?- neq(A,B), A = f(C), B = f(D).
neq(f(C),f(D))
domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <->
    intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...

Using Reactivation for Finite Domains
domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>  
    intersection(L1,L2,L3), domain(X,L).
intersection(L1,L2,L3) :- ...

?- domain(X,[dog,fox,horse,snails,zebra]).
Using Reactivation for Finite Domains

domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2) <=>
  intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...

?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
domain(X,L)  <->  ground(X) | memberchk(X,L).
domain(X,[])  <->  fail.
domain(X,[V])  <->  X = V.
domain(X,L1), domain(X,L2)  <=>
    intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...

?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
Using Reactivation for Finite Domains

\begin{verbatim}
domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>  
    intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...
\end{verbatim}

?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>  intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...

?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
Using Reactivation for Finite Domains

domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>
    intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...

?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snails,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
domain(X,[s,z])
Using Reactivation for Finite Domains

domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2) <=>
  intersection(L1,L2,L), domain(X,L).
intersection(L1,L2,L3) :- ...

?- domain(X,[dog,fox,horse,snails,zebra]).
domain(X,[dog,fox,horse,snail,zebra])
?- domain(X,[d,f,h,s,z]), X = z.
  X = z
?- domain(X,[d,f,h,s,z]), domain(X,[c,s,z]).
domain(X,[s,z])
?- domain(X,[d,f,h,s,z]), domain(X,[c,z]).
Using Reactivation for Finite Domains

\[
domain(X,L) \iff \text{ground}(X) \lor \text{memberchk}(X,L).
\]
\[
domain(X,[]) \iff \text{fail}.
\]
\[
domain(X,[V]) \iff X = V.
\]
\[
domain(X,L1), \ domain(X,L2) \iff 
\text{intersection}(L1,L2,L), \ domain(X,L).
\]
\[
\text{intersection}(L1,L2,L3) :- \ ... 
\]

?- \ domain(X,[\text{dog,fox,horse,snails,zebra}]).
\text{domain}(X,[\text{dog,fox,horse,snails,zebra}])

?- \ domain(X,[\text{d,f,h,s,z}]), \ X = z.
\]
\text{X} = z

?- \ domain(X,[\text{d,f,h,s,z}]), \ domain(X,[c,s,z]).
\text{domain}(X,[s,z])

?- \ domain(X,[\text{d,f,h,s,z}]), \ domain(X,[c,z]).
\text{X} = z
Reactivation

- rules may not fire because of lack of instantiation
Summary

Reactivation

- rules may not fire because of lack of instantiation
- upon instantiation, the rule may now fire
Summary

Reactivation

▶ rules may not fire because of lack of instantiation
▶ upon instantiation, the rule may now fire
▶ useful for implementing constraint solvers
1. Introduction
2. The Art of CHR
3. Simpagation
4. For the Sake of Arguments
5. En Guarde
6. A Perfect Match
7. Bigger Programs
8. Propagation
9. Order in the Rules
10. Reactivation
11. CHR vs. Prolog
12. Tutorial Summary
13. Facts about CHR
## Summary: CHR vs Prolog

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>CHR</th>
</tr>
</thead>
<tbody>
<tr>
<td>heads</td>
<td>1</td>
<td>≥ 1</td>
</tr>
<tr>
<td>rule selection</td>
<td>unification</td>
<td>matching &amp; guard</td>
</tr>
<tr>
<td>different rules</td>
<td>alternatives/backtracking</td>
<td>try all in sequence</td>
</tr>
<tr>
<td>no rule</td>
<td>failure</td>
<td>delay</td>
</tr>
</tbody>
</table>
CHR and Backtracking

\begin{verbatim}
\text{p :- a. \ a <=> c1. \ b <=> d1.}
\text{p :- b. \ a <=> c2. \ b <=> d2.}
\end{verbatim}

\text{?- p.}

\text{c1}
CHR and Backtracking

p :- a. a <-> c1. b <-> d1.
p :- b. a <-> c2. b <-> d2.

?- p.
c1 ; y
d1
CHR and Backtracking

p :- a. a <=> c1. b <=> d1.
p :- b. a <=> c2. b <=> d2.

?- p.
c1 ; y
d1

- Prolog creates choicepoints
- CHR does not
- Prolog backtracking undoes CHR changes
Backtracking for Labeling

\[
\text{domain}(X,L) \iff \text{ground}(X) \mid \text{memberchk}(X,L).
\]
\[
\text{domain}(X,[]) \iff \text{fail}.
\]
\[
\text{domain}(X,[V]) \iff X = V.
\]
\[
\text{domain}(X,L1), \text{domain}(X,L2) \iff
\begin{align*}
\text{intersection}(L1,L2,L), & \quad \text{domain}(X,L).
\end{align*}
\]
\[
\text{split}(X) \iff \text{ground}(X) \mid \text{true}.
\]
\[
\text{split}(X), \text{domain}(X,[V|Vs]) \iff X = V \mid \text{domain}(X,Vs).
\]

?- \text{domain}(X,[d,z]), \text{split}(X).

Backtracking for Labeling

domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>  
    intersection(L1,L2,L), domain(X,L).

split(X)  <=>  ground(X) | true.
split(X), domain(X,[V|Vs])  <=>  X = V ; domain(X,Vs).

?- domain(X,[d,z]), split(X).
X = d  ;  y
X = z
Backtracking for Labeling

\[
\text{domain}(X,L) \iff \text{ground}(X) \mid \text{memberchk}(X,L).
\]

\[
\text{domain}(X,[]) \iff \text{fail}.
\]

\[
\text{domain}(X,[V]) \iff X = V.
\]

\[
\text{domain}(X,L1), \text{domain}(X,L2) \iff \\
\quad \text{intersection}(L1,L2,L), \text{domain}(X,L).
\]

\[
\text{split}(X) \iff \text{ground}(X) \mid \text{true}.
\]

\[
\text{split}(X), \text{domain}(X,[V|Vs]) \iff X = V \mid \text{domain}(X,Vs).
\]

?- \text{domain}(X,[d,z]), \text{split}(X).
X = d \quad ; \quad y
X = z

?- \text{domain}(X,[d,f,z]), \text{split}(X).
Backtracking for Labeling

domain(X,L) <=> ground(X) | memberchk(X,L).
domain(X,[]) <=> fail.
domain(X,[V]) <=> X = V.
domain(X,L1), domain(X,L2) <=>
    intersection(L1,L2,L), domain(X,L).

split(X) <=> ground(X) | true.
split(X), domain(X,[V|Vs]) <=> X = V ; domain(X,Vs).

?- domain(X,[d,z]), split(X).
X = d ; y
X = z

?- domain(X,[d,f,z]), split(X).
X = d ; y
domain(X,[f,z])
domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>
    intersection(L1,L2,L), domain(X,L).

indomain(X)  <=>  ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn])  <=>  X = V1;...;X = Vn.
Backtracking for Labeling

domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>
    intersection(L1,L2,L), domain(X,L).

indomain(X)  <=>  ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn])  <=>  X = V1;...;X = Vn.
indomain(X), domain(X,Vs)  <=>  member(X,Vs).
domain(X,L)  \iff \text{ground}(X) \text{ or } \text{memberchk}(X,L).
domain(X,[])  \iff \text{fail}.
domain(X,[V])  \iff X = V.
domain(X,L1), domain(X,L2) \iff
\text{intersection}(L1,L2,L), \text{domain}(X,L).

\text{indomain}(X)  \iff \text{ground}(X) \text{ or } \text{true}.
\% \text{indomain}(X), \text{domain}(X,[V1,...,Vn])  \iff X = V1;...;X = Vn.
\text{indomain}(X), \text{domain}(X,Vs)  \iff \text{member}(X,Vs).

?- \text{domain}(X,[d,f,z]), \text{indomain}(X).
Backtracking for Labeling

domain(X,L)  <=>  ground(X) | memberchk(X,L).
domain(X,[])  <=>  fail.
domain(X,[V])  <=>  X = V.
domain(X,L1), domain(X,L2)  <=>
    intersection(L1,L2,L), domain(X,L).

indomain(X)  <=>  ground(X) | true.
% indomain(X), domain(X,[V1,...,Vn])  <=>  X = V1;...;X = Vn.
indomain(X), domain(X,Vs)  <=>  member(X,Vs).

?- domain(X,[d,f,z]), indomain(X).
X = d  ; y
X = f  ; y
X = z
You should now have an understanding of:
You should now have an understanding of:

- how CHR works.
You should now have an understanding of:

- how CHR works.
- how CHR differs from Prolog.
You should now have an understanding of:

- how CHR works.
- how CHR differs from Prolog.
- that CHR is not exclusively about constraints.
**History: Programming Highlights**

**1991** **CHR** is born, Thom Frühwirth
1991  **CHR** is born, Thom Frühwirth

1995  Christian Holzbaur implements **CHR(SICStus)**
1991  **CHR** is born, Thom Frühwirth
1995  Christian Holzbaur implements **CHR(SICStus)**
2002  **Leuven CHR** is born
1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

2002-2005 optimized compilation & program analysis (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers
1991   **CHR** is born, Thom Frühwirth

1995   Christian Holzbaur implements **CHR(SICStus)**

2002   **Leuven CHR** is born

2002-2005  **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers

2004   **refined semantics**, Gregory Duck et al.
1991  CHR is born, Thom Frühwirth
1995  Christian Holzbaur implements CHR(SICStus)
2002  Leuven CHR is born
2002-2005  optimized compilation & program analysis (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers
2004  refined semantics, Gregory Duck et al.
2004  1st CHR workshop
1991 **CHR** is born, Thom Frühwirth
1995 Christian Holzbaur implements **CHR(SICStus)**
2002 **Leuven CHR** is born
2002-2005 **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers
2004 **refined semantics**, Gregory Duck et al.
2004 1st **CHR workshop**
2005-2008 **computational complexity**, PhD Jon Sneyers
1991 CHR is born, Thom Frühwirth
1995 Christian Holzbaur implements CHR(SICStus)
2002 Leuven CHR is born
2002-2005 optimized compilation & program analysis (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers
2004 refined semantics, Gregory Duck et al.
2004 1st CHR workshop
2005-2008 computational complexity, PhD Jon Sneyers
2005- Leuven JCHR (Java), Peter Van Weert
1991 **CHR** is born, Thom Frühwirth

1995 Christian Holzbaur implements **CHR(SICStus)**

2002 **Leuven CHR** is born

2002-2005 **optimized compilation & program analysis** (abstract interpretation), PhDs of Gregory Duck and Tom Schrijvers

2004 **refined semantics**, Gregory Duck et al.

2004 1st **CHR workshop**

2005-2008 **computational complexity**, PhD Jon Sneyers

2005- **Leuven JCHR** (Java), Peter Van Weert

2007 first **concurrent system**, Sulzmann & Lam
CHR Systems

Logic Programming languages

Now let's consider CHRs and their applications in the context of functional programming.

**4.1.2 CHR(FP)**

As a form of concurrent constraint programming, CHRs have been successfully applied in the context of Functional Programming, as shown in Section 7.3.1. Several CHRs were developed specifically for this purpose. The most noteworthy is the Chameleon system (Stuckey and Sulzmann 2005), which uses CHRs as its programming language for an extensible type system. Internally, Chameleon uses the HaskellCHR implementation.
CHR Programs

Thom Frühwirth, with his students:
- lots of example programs, constraint solvers and others

Thom Frühwirth & Tom Schrijvers,
  - union-find

Jon Sneyers,
  - Fibonacci heaps and Dijkstra’s shortest path
  - Hopcroft’s DFA minimization
  - Turing and RAM machine simulators

Henning Christiansen,
  - meta-programming in CHR

...
Declarative Semantics

- classical first-order logic (Frühwirth)
- linear logic (Betz)
CHR Theory & Applications

Declarative Semantics
- classical first-order logic (Frühwirth)
- linear logic (Betz)

Rewriting Properties
- confluence (Frühwirth; Duck; Haemmerle)
- completion (Frühwirth & Abdennadher)
- termination (Frühwirth; Voets & Pilozzi)
- complexity (Frühwirth; De Koninck)

Applications
- type systems (Sulzmann & Stuckey)
- test case generation (Schrijvers)
- multi-agent systems (Alberti)
Declarative Semantics

- classical first-order logic (Frühwirth)
- linear logic (Betz)

Rewriting Properties

- confluence (Frühwirth; Duck; Haemmerle)
- completion (Frühwirth & Abdennadher)
- termination (Frühwirth; Voets & Pilozzi)
- complexity (Frühwirth; De Koninck)

Applications

- type systems (Sulzmann & Stuckey)
- test case generation (Schrijvers)
- multi-agent systems (Alberti)
- ...
CHR Researchers around the World

1. Ulm, Germany (Frühwirth, Meister, Betz, Djelloul, Raiser, ...)
2. ... while the Ulm group investigates alternative logical semantics for CHR (Section 2.1).
3. The study of an implementation of CHR and other systems.
4. In 2003, the Ulm group leads the creation of a CHR system.
5. In 2006, the Ulm group starts a Ph.D. program.
6. In 2007, the Ulm group starts working on the CHR system.
7. In 2015, the Ulm group starts working on the CHR system.
8. In 2005, the Ulm group leads the creation of a CHR system.
9. In 2006, the Ulm group starts working on the CHR system.
10. In 2001, the Ulm group leads the creation of a CHR system.
11. In 1998, the Ulm group leads the creation of a CHR system.
12. In 2000, the Ulm group leads the creation of a CHR system.
13. In 1999, the Ulm group leads the creation of a CHR system.
14. In 2002, the Ulm group leads the creation of a CHR system.
15. In 2004, the Ulm group leads the creation of a CHR system.
CHR in Industry

- multi-headed business rules
- custom constraint solvers
multi-headed business rules

- Scientific Software & Systems Ltd.
  - stock brokering software
- Cornerstone Technology Inc
  - injection mould design tool
- BSSE System and Software Engineering
  - test generation
- MITRE Corporation
  - optical network design
Further Reading

- CHR website (Google for)
  - programs
  - papers
- CHR Survey
Thank You!

My Questions: Get the Quiz!

Your Questions?