An introduction to Constraint Handling Rules

Jon Sneyers
September 2011

PART ONE

Introduction
How to get from A to B?

point A: Universiteitshallen
Naamsestraat 22, Leuven, Belgium

point B: Library
Ladeuzeplein, Leuven

shortest path (by car) according to Google Maps
How to find the shortest path???

Edsger Dijkstra (1930-2002)
Dutch computer scientist
Dijkstra's algorithm:

1. distance(start-point) = 0
2. pick a (not-yet-considered) point \( x \) with smallest distance, \( \text{LABEL}(x) \)
3. if end-point is considered, stop; otherwise go to step 2

\( \text{LABEL}(x) \): for all arrows \( x \xrightarrow{a} y \):
set distance(y) = distance(x) + a
(if the new distance is shorter)

Edsger Dijkstra (1930-2002)
Dutch computer scientist

How to do this efficiently?
Implementing Dijkstra's algorithm

...in machine code

and so on...

...in assembly

and so on...
Implementing Dijkstra's algorithm

**In C**

```c
void insert_min_key (node *, node *, int, float *);
void deletenode (node *); // delkey from heap
void minkey (node *, float *); // get key of minnode
```

**In Java**

```java
public final class DijkstraShortestPathV2, E {
  private List<E> edgeList;
  private double length;
  public DijkstraShortestPathV2 (GraphV, V graph, V startVertex, V endVertex) { /* implementation */ }
  public DijkstraShortestPathV2 (GraphV, V graph, V startVertex, V endVertex, double radius) { /* implementation */ }
  public AStarShortestPathV2 (GraphV, V graph, V startVertex, V endVertex) { /* implementation */ }
  public Path getShortestPath (GraphV, V graph, V startVertex, V endVertex) { /* implementation */ }
  return pathList;
}
```
Implementing Dijkstra's algorithm

\[\text{dijkstra}(\text{Vertex}, Ss) :\]
\[
\text{create}(\text{Vertex}, [\text{Vertex}], Ds),
\text{dijkstra}(\text{1Ds}, [\text{Vertex}(0,0)], Ss).
\]
\[
\text{dijkstra}_1([], Ss, Ss).
\]
\[
\text{dijkstra}_1([D|Ds], Ss0, Ss) :\]
\[
\text{best}(Ds, D, S),
\text{delete}(D|Ds, [S], Ds1),
Ss1(\text{Vertex}, \text{Distance}, \text{Path}),
\text{reverse}([\text{Vertex}, \text{Path}], \text{Path1}),
\text{merge}(Ss0, Ss1, [\text{Vertex}, \text{Distance}, \text{Path1}], \text{Ss1}),
\text{create}(\text{Vertex}, [\text{Vertex}, \text{Path}], Ds2),
\text{delete}(Ds2, Ss1, Ds3),
\text{inc}(Ds3, \text{Distance}, Ds4),
\text{merge}(Ds4, Ds, Ds5),
\text{dijkstra}_1([Ds5], Ss, Ss).
\]
\[
\text{path}(\text{Vertex}, \text{Vertex}, \text{Path}, \text{Dist}) :\]
\[
\text{dijkstra}_1([Ss], Ss), !.
\]
\[
\text{create}(\text{Start}, \text{Path}, \text{Edges}) :\]
\[
\text{setof}([\text{Start}, \text{Vertex}, \text{Edge}], \text{Edges}, E),
\text{e}(\text{Start}, \text{Vertex}, \text{Edge}, \text{Edges}), !.
\]
\[
\text{create}(\_, _, []) !.
\]
\[
\text{best}([], \text{Best}, \text{Best}) !.
\]
\[
\text{best}([\text{Edge}\text{Edges}], \text{Best0}, \text{Best}) :\]
\[
\text{edge}(\text{Edge}, \text{Best0}), !.
\]
\[
\text{best}([\text{Edges}], \text{Best}, \text{Best0}) !.
\]
\[
\text{shorter}(s, \_X, \_Y) :\]
\[
X < Y.
\]
\[
\text{delete}(\_, [\_], \_).
\]
\[
\text{delete}(\_Xs, [\_ | \_Xs]), !.
\]
\[
\text{eq}(X, Y, !).
\]
\[
\text{eq}(X, Y, !).
\]

...in Prolog

Implementing Dijkstra's algorithm

\[\text{dijkstra}(A) \iff \text{scan}(A, B),
\text{scan}(L, \text{edge}(\text{N}, \text{W})) \iff \text{LSL} \iff \text{relabel}(\text{N}, \text{L}),
\text{relabel}(\text{L}, \text{W}).
\]
\[
\text{dijkstra}(L) \iff \text{distance}(\text{N}, \text{W}),
\text{execute} \text{min}(\text{N}, \text{L}),
\text{distance}(\text{N}, \text{W}) \iff \text{relabel}(\text{N}, \text{W}).
\]
\[
\text{execute} \text{min}(\text{N}, \text{L}).
\]

...in CHR

\[
\text{findmin} \iff \text{item}(I, K, \_), \text{K} = 0, \text{K} = 0.
\]
\[
\text{delete}(\text{K}, \text{I}) \iff \text{K} = \text{K} + 1.
\]
\[
\text{ch2rt}(I) \iff \text{item}(I, K, R, u, \_).
\]
\[
\text{extract}_\text{min}(\text{X}, \text{Y}, \text{K}, \text{I}) \iff \text{ch2rt}(I), \text{K} = \text{I}.
\]
\[
\text{ch2rt}(I) \iff \text{true}.
\]
\[
\text{mark}(\text{I}) \iff \text{item}(\text{I}, \text{K}, \text{R}, \text{P}, \text{M}).
\]
\[
\text{mark}(\text{I}) \iff \text{K} < 0, \text{K} = \text{K}.
\]
\[
\text{mark}(\text{I}) \iff \text{mark}(\text{I}, \text{K}, \text{R}, \text{P}, \text{M}).
\]
\[
\text{mark}(\text{I}) \iff \text{mark}(\text{I}, \text{K}, \text{R}, \text{P}, \text{M}).
\]

...in CHR
Implementing Dijkstra's algorithm

```
:- chr_constraint edge(+node,+node,+length),
     source(+node),
     distance(+node,+length).
:- chr_type node == int.
:- chr_type length == number.
1 :: source(V) ==> distance(V,0).
1 :: distance(V,D1) \ distance(V,D2) <=> D1 <= D2 | true.
D+2 :: distance(V,D), edge(V,C,W) ==> distance(W,D+C).
```

...in CHR

CHR = Constraint Handling Rules

- CHR is a very high level programming language
- based on rules
  - propagation rules:
    - clouds ⇒ forecast(rainy).
    - forecast(rainy) ⇒ bring(coat).
    - forecast(sunny) ⇒ bring(sunscren).
  - simplification rules:
    - bring(coat), bring(sunscren) ⇔ bring(umbrella).
- stand-alone (CHR-only)
  or extending a host language
**CHR = Constraint Handling Rules**

- CHR is a very **high level** programming language
- based on **patience**
  - propagation rules:
    - clouds \(\Rightarrow\) forecastedly.
    - bring(coat) \(\Rightarrow\) bring(umbrella).
- **simplification rules**:
  - bring(coat), bring(sunscreen) \(\Leftrightarrow\) bring(umbrella).
- stand-alone (CHR-only)
- or extending a **host language**

**Syntax of CHR**

- **Propagation rule:**
  \[
  \text{head} \Rightarrow \text{guard} \mid \text{body}.
  \]

  **Example:** \(\text{dist}(A,D), \text{road}(A,B,L) \Rightarrow \text{dist}(B,D+L).\)
  
  *"If A is within distance D, and there is a road from A to B of length L, then B is within distance D+L."*

- **Simplification rule:**
  \[
  \text{head} \Leftrightarrow \text{guard} \mid \text{body}.
  \]

  **Example:** \(\text{dist}(A,X), \text{dist}(A,Y) \Leftrightarrow X \leq Y \mid \text{dist}(A,X).\)
  
  *"If A is within distance X and also within distance Y, where \(X \leq Y\), then this is equivalent to A being within distance X."*
Operational semantics of CHR

IF head IN STORE (AND guard HOLDS), THEN...

- **Propagation rule:** ... ADD body TO STORE
  - Keep head, add body
    \[
    \text{head} \Rightarrow \text{guard} \mid \text{body}.
    \]

- **Simplification rule:** ... REPLACE head BY body
  - Remove head, add body
    \[
    \text{head} \Leftarrow \text{guard} \mid \text{body}.
    \]

- “Simpagation” rule combines the above:
  - Remove removed_head, keep kept_head, add body
    \[
    \text{kept}_\text{head} \setminus \text{removed}_\text{head} \Leftarrow \text{guard} \mid \text{body}.
    \]
Features of CHR

- Embedded in a host language:
  CHR extends an existing programming language, e.g.
  - CHR(Prolog)
  - CHR(Haskell)
  - CHR(Java)
  - CHR(C)

- Simplification rule:
  head \iff guard \lor body.

  Example: \( \text{dist}(A,X), \text{dist}(A,Y) \iff X \leq Y \lor \text{dist}(A,X) \).

  "If A is within distance X and also within distance Y, where \( X \leq Y \), then this is equivalent to A being within distance X."

- Propagation rule:
  head \implies guard \lor body.

  Example: \( \text{dist}(A,D), \text{road}(A,B,L) \implies \text{dist}(B,D+L) \).

  "If A is within distance D, and there is a road from A to B of length L, then B is within distance D+L."

- Simplification rule:
  head \iff guard \lor body.

  Example: \( \text{dist}(A,X), \text{dist}(A,Y) \iff X \leq Y \lor \text{dist}(A,X) \).

  "If A is within distance X and also within distance Y, where \( X \leq Y \), then this is equivalent to A being within distance X."

- Multiple heads:

  The head of a rule consists of an arbitrary number of CHR constraints (1 or more)

  cf. Prolog: single-headed
Features of CHR

- Propagation rule:
  \[
  \text{head} \Rightarrow \text{guard} \mid \text{body}.
  \]
  Example: \(\text{dist}(A,D), \text{road}(A,B,L) \Rightarrow \text{dist}(B,D+L).
  \)
  “If A is within distance D, and there is a road from A to B of length L, then B is within distance \(D+L\).”

- Simplification rule:
  \[
  \text{head} \Leftrightarrow \text{guard} \mid \text{body}.
  \]
  Example: \(\text{dist}(A,X), \text{dist}(A,Y) \Leftrightarrow X \leq Y \mid \text{dist}(A,X).
  \)
  “If A is within distance \(X\) and also within distance \(Y\), where \(X \leq Y\), then this is equivalent to A being within distance \(X\).”

Multi-set semantics
The constraint store may contain the same constraint multiple times.
\(\{c\} \) is not the same as \(\{c,c\}\).

\(\text{cf. classical logic: } p \leftrightarrow p \land p\)

Important remark:
in CHR(Prolog), we can still use Prolog disjunction or nondeterministic predicates in the \textbf{body} of rules!
CHR with disjunction/search is called \textbf{CHR}.

Committed-choice
Once a rule has been applied, it remains applied – no backtracking to try different derivation paths.

\(\text{cf. Prolog: choice-points and backtracking}\)

\[
\text{head} \Leftrightarrow \text{guard} \mid \text{body}.
\]

Example: \(\text{collect}(A), \text{element}(X) \Leftrightarrow \text{collect}([X|A])\).
“Collect the elements in a list.”
\(\text{element}(a), \text{element}(b), \text{element}(c), \text{collect}([]) \rightarrow \text{collect}([a,b,c])\).
Features of CHR

- **Propagation rule:**
  \[ \text{head} \implies \text{guard} \mid \text{body}. \]

Example: \( \text{dist}(A,D), \text{road}(A,B,L) \implies \text{dist}(B,D+L). \)

“If A is within distance D, and there is a road from A to B of length L, then B is within distance D+L.”

- **Simplification rule:**
  \[ \text{head} \iff \text{guard} \mid \text{body}. \]

Example: \( \text{dist}(A,X), \text{dist}(A,Y) \iff X \leq Y \mid \text{dist}(A,X). \)

“If A is within distance X and also within distance Y, where \( X \leq Y \), then this is equivalent to A being within distance X.”

---

**Logical semantics**

CHR has a declarative semantics!

---

**PART TWO**

Writing CHR programs
CHR(Prolog) by example

- Simple example: color mixing in CHR
- We first declare CHR constraints as follows:
  \[- \text{chr\_constraint red, blue, yellow, purple, green, orange.}\]

- Then we write the rules:
  
  \[
  \text{red, blue } \leftrightarrow \text{ purple.} \\
  \text{blue, yellow } \leftrightarrow \text{ green.} \\
  \text{yellow, red } \leftrightarrow \text{ orange.}
  \]

CHR(Prolog) by example

- Simple example: color mixing in CHR
  
  red, blue \leftrightarrow purple.
  blue, yellow \leftrightarrow green.
  yellow, red \leftrightarrow orange.

- CHR program execution:
  
  - user gives a \textit{goal}
  - rules are applied exhaustively
  - the remaining constraints are the \textit{result}
CHR(Prolog) by example

- Simple example: color map
  - red, blue \(\leftrightarrow\) purple
  - blue, yellow \(\leftrightarrow\) green
  - yellow, red \(\leftrightarrow\) orange

- Example interaction:
  - ?- blue, red.
  - purple
  - ?- yellow, blue, red.

  green
  red

Why this answer?
(and not, say, "yellow, purple")

Refined semantics
Execution from left to right and from top to bottom (cf. Prolog)

Confluence

- Simple example: color map in CHR
  - r1 @ red
  - r2 @ blue
  - r3 @ yellow, red \(\leftrightarrow\) orange

  ?- yellow, blue, red.

  green
  red
  purple
  yellow
  blue

Abstract semantics
allows rule application in any order
Confluence

A CHR program is called **confluent** if for any given goal, there is only one result, regardless of the order in which rules are applied. *(so the color mixing program is not confluent)*

**Abstract semantics**
allows rule application in any order

- ?- yellow, blue, red.

  green
  red
  result 1

  purple
  yellow
  result 2

  orange
  blue
  result 3

Constraints with arguments

- Add anything to brown and it remains brown:
  
  red, blue <=> purple.
  blue, yellow <=> green.
  yellow, red <=> orange.
  
  brown, red <=> brown.
  brown, blue <=> brown.
  brown, yellow <=> brown.
  brown, purple <=> brown.
  ...
Constraints with arguments

- From many 0-ary constraints to one unary constraint:

```prolog
:- chr_constraint red, blue, yellow, purple, green, orange.
red, blue  <->  purple.
blue, yellow  <->  green.
yellow, red  <->  orange.
```

```prolog
:- chr_constraint color/1.
color(red), color(blue)  <->  color(purple).
...```

- Now we can write more general rules:

```prolog
:- chr_constraint color/1.
color(X), color(Y)  <->  mix(X,Y,Z)  |  color(Z).
color(brown), color(_)  <->  color(brown).
```

% host language

```prolog
mix(red,blue,purple).
mix(blue,yellow,green).
mix(yellow,red,orange).
```
Type and mode declarations

- Optionally, we can specify types and modes:

  % no type/mode declaration:
  :- chr_constraint color/1.

  % only mode declaration:
  :- chr_constraint color(+). % ground argument

  % type and mode declaration:
  :- chr_constraint color(+colorname).
  :- chr_type colorname --->
     red ; blue ; yellow ; purple ; green ; orange.

Simpagation rules

- So far we have only used simplification rules.
- Simpagation rules can be more concise/efficient:

  % simplification rule:
  color(brown), color(_) <=> color(brown).

  “true”?  
  In Prolog, “true” is a built-in that does not do anything. We use it to indicate an empty body.

  % simpagation rule:
  color(brown) \ color(_) <=> true.
Typical pattern #1: flattening lists

- We want to convert "colors([red, green, blue])" to "color(red), color(green), color(blue)"

```prolog
:- chr_constraint color(+colorname).
:- chr_type colorname ---> red ; blue ; yellow ;...
:- chr_constraint colors(+list(colorname)).
:- chr_type list(T) ---> [] ; [T|list(T)].

colors([]) <=> true.
colors([C|Rest]) <=> color(C), colors(Rest).
```

(just like how you would do this in Prolog)

Typical pattern #2: "default constructor"

- Now we have not a fixed quantity of paint, but we specify the amount
- For backwards compatibility, we still have color/1

```prolog
:- chr_constraint color(+colorname).
:- chr_constraint color(+colorname,+amount).
:- chr_type colorname ---> red ; blue ; yellow ;...
:- chr_type amount == float.

% we assume 1 liter of paint:
color(C) <=> color(C,1).
```
Typical pattern #3: maintaining a sum

:- chr_constraint color(+colorname,+amount).

color(C,A1), color(C,A2)
    <=> TA is A1+A2, color(C,TA).

color(C,0) <=> true.
color(X,A1), color(Y,A2)
    <=> mix(X,Y,Z) | TA is A1+A2,
        color(Z,TA).

Typical pattern #4: maximum

- Which color do we have the most of?

:- chr_constraint color(+colorname,+amount).
:- chr_constraint most(+colorname).

color(C,A) ==> most(C,A).
most(_,A1) \ most(_,A2) <=> A1 >= A2 | true.
CHR(Prolog) one-liners (1)

- Finding the minimum:

\[
\text{min}(A) \land \text{min}(B) \iff A \leq B \lor \text{true}.
\]

?- \text{min}(8), \text{min}(3), \text{min}(6), \text{min}(7).
\text{min}(3)

- Computing the sum:

\[
\text{sum}(A), \text{sum}(B) \iff C \text{ is } A+B, \text{sum}(C).
\]

?- \text{sum}(3), \text{sum}(5), \text{sum}(6).
\text{sum}(14)
CHR(Prolog) one-liners (2)

- Transitive closure

```prolog
:- op(700,xfx,before).
:- chr_constraint before(+any,+any).

A before B, B before C ==> A before C.
```

?- a before b, b before c, c before d.
a before b
b before c
c before d
a before c
a before d
b before d

CHR(Prolog) one-liners (3)

- Naive merge-sort in $O(n^2)$ time

```prolog
:- op(700,xfx,before).
:- chr_constraint before(+any,+any).

A before B \ A before C

\<=>\ B @< C | B before C.
```

?- 0 before foo, 0 before bar, 0 before baz, 0 before quux.
0 before bar
bar before baz
baz before foo
foo before quux
CHR(Prolog) two-liners (1)

- Greatest common divisor
  (Euclid's algorithm)

```prolog
:- chr_constraint gcd(+int).

gcd(0) <=> true.

gcd(N) \ gcd(M) <=> N =< M |
            L is M mod N, gcd(L).
```

?- gcd(94017), gcd(1155), gcd(2035).
gcd(11)

CHR(Prolog) two-liners (2)

- Prime number generator
  (sieve of Eratosthenes)

```prolog
:- chr_constraint prime(+int).

prime(N) ==> N>2 | M is N-1, prime(M).

prime(A) \ prime(B) <=> B mod A =:= 0 | true.
```

?- prime(10).
prime(2)
prime(3)
prime(5)
prime(7)
CHR(Prolog) two-liners (3)

- Fibonacci numbers

```prolog
:- chr_constraint fib(+int,+int), upto(+int).

upto(_) ==> fib(0,1), fib(1,1).

upto(Max), fib(N1,M1), fib(N2,M2)
    ==> Max>N2, N2 is N1+1 | 
        N is N2+1, M is M1+M2, fib(N,M).
```

?- upto(10).
   fib(10,89)
   fib(9,55)
   fib(8,34)
   fib(7,21)
   fib(6,13)
   ...

CHR(Prolog) two-liners (4)

- Optimal merge-sort

```prolog
:- op(700,xfx,before).
:- chr_constraint before(+any,+any), sort(+int,+any).

X before A \ X before B <=> A @< B |
       A before B.

sort(N,A), sort(N,B) <=> A @< B |
    M is N+1, sort(M,A), A before B.
```

?- sort(0,foo), sort(0,bar), sort(0,baz), sort(0,quux).
   bar before baz
   baz before foo
   foo before quux
   sort(2,bar)
CHR(Prolog) two-liners (5)

- Soduko puzzle solver in CHR

```prolog
:- chr_constraint given(+pos,+val), maybe(+pos,list(val)).

given(P1,V) \ maybe(P2,L) <=> sees(P1,P2), select(V,L,R) | maybe(P2,R).

maybe(P,L) <=> member(V,L), given(P,V).

sees(X-_, X-_).
sees(_,X, _-X).
```

?- given(a-1,5), given(f-4,3), ..., maybe(a-2, [1,2,3,...,9]), ...
given(a-1,5)
given(a-2,2)
given(a-3,7)
...

One last example...

- Simple less-than-or-equal constraint solver

```prolog
:- op(700,xfx,leq).
:- chr_constraint leq/2.

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

?- A leq B, B leq C, C leq A.
A = B
B = C
Differences between CHR and Prolog

<table>
<thead>
<tr>
<th></th>
<th>Prolog</th>
<th>CHR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>basic elements</strong></td>
<td>predicates</td>
<td>constraints</td>
</tr>
<tr>
<td><strong>elements are defined by</strong></td>
<td>clauses</td>
<td>rules</td>
</tr>
<tr>
<td><strong>syntax</strong></td>
<td>head :- body.</td>
<td>head &lt;=&gt; guard</td>
</tr>
<tr>
<td><strong>#heads</strong></td>
<td>1</td>
<td>1, 2, 3, ...</td>
</tr>
<tr>
<td><strong>definition selection condition</strong></td>
<td>unification</td>
<td>matching + guard</td>
</tr>
<tr>
<td><strong>different applicable definitions</strong></td>
<td>try alternatives (backtracking)</td>
<td>committed-choice</td>
</tr>
<tr>
<td><strong>no applicable definition</strong></td>
<td>failure</td>
<td>suspension (delay) ↓ partial result</td>
</tr>
</tbody>
</table>

Committed-choice – different from Prolog!

- In Prolog, **backtracking** (proof search) is used to find a non-failing derivation
- In CHR there is no backtracking

```prolog
:- chr_constraint chr/0, output/1.
chr <=> output(foo).
chr <=> output(bar).
prolog :- output(foo).
prolog :- output(bar).

?- prolog.
output(foo) ;
?- chr.
output(foo)
```

output(bar)
Head matching – different from Prolog!

- In Prolog, **unification** is used to match clause heads
- In CHR, **matching** (one-way unification) is used

```prolog
:- chr_constraint chr/1, output/1.
    chr(foo) <=> output(bar).
    prolog(foo) :- output(bar).
```

```
?- prolog(foo).
output(bar)

?- prolog(Variable).
output(bar)
Variable = foo

?- prolog(quux).
No

?- chr(foo).
output(bar)

?- chr(Variable).
chr(Variable)

?- chr(quux).
chr(quux)
```

### PART THREE

**Theory & Applications**
History of CHR: some milestones
(not including applications)

1991 CHR is born, Thom Frühwirth
1993 First CHR compiler by Pascal Brisset
1995 Christian Holzbaur implements CHR(SICStus)
1998 Confluence, program analysis (PhD Abdennadher)
2002 Tom Schrijvers implements Leuven CHR system
   Optimizing compilation (PhDs Duck, Schrijvers, Sneyers, Van Weert)
2003 First CHR book [Frühwirth & Abdennadher, Essentials of Constraint Programming]
2004 Refined semantics formalized [Duck et al.]
   First CHR workshop (Ulm, Germany)
2005 Peter Van Weert implements Leuven JCHR (Java)
2007 Sulzmann & Lam implement first concurrent system
2009 Second CHR book [Frühwirth, Constraint Handling Rules]
2010 First CHR summer school (Leuven, Belgium)
2011 Second summer school, 8th workshop (Cairo, Egypt)
Theory topics (1)

- Semantics
  - Declarative (logical) semantics
    - Classical logic (Frühwirth)
    - Linear logic (Betz)
    - Transaction logic, …
  - Operational semantics
    - Abstract semantics
    - Theoretical semantics
    - Refined semantics (Duck et al)
    - Priority semantics (De Koninck)

Theory topics (2)

- Relationship to other formalisms
  - Term rewriting (ACD term rewriting, Duck, Stuckey et al)
  - Production rules / business rules (Van Weert)
  - Join-Calculus (Sulzmann and Lam)
  - Logical Algorithms (De Koninck)
  - Graph Transformation Systems (Raiser)
  - Petri nets (Betz)
  - …
Theory topics (3)

- Program analysis
  - Confluence (Abdennadher, Duck et al, Raiser&Tacchella, Haemmerlé&Fages, …)
  - Operational equivalence (Abdennadher&Frühwirth)
  - Termination (Frühwirth, Pilozzi, Voets)
  - Complexity (Frühwirth&Schrijvers, Sneyers, De Koninck)
  - Abstract interpretation (Schrijvers, Stuckey, Duck)
  - ...

Theory topics (4)

- Extensions and variants of CHR
  - Disjunction / Search (Abdennadher, Wolf, De Koninck)
  - Negation, aggregates (Van Weert, Sneyers)
  - Modularity (e.g. CHRat)
  - Adaptive CHR (Wolf et al)
  - Probabilistic CHR (e.g. CHRiSM)
  - ...

Application domains

- Constraint solvers
  - CHR was specifically designed for this
  - Some domains where CHR has been used:
    - Scheduling
    - Soft constraints
    - Spatio-temporal reasoning
    - Multi-agent systems
    - Semantic web

- General-purpose programming language
  - Many classical algorithms have been implemented in CHR in a very elegant and natural way - often more concise than pseudocode!

Application domains

- Programming language development
  - Type systems (e.g. Haskell type classes)
  - Abductive reasoning
  - Computational linguistics (NLP)
    - CHR Grammars (Dahl&Christiansen)
  - Meta-programming
  - Testing & verification

- CHR can be used as a high-performance business rule engine (integrated in your favorite host language!)
Good starting points

  [http://www.constraint-handling-rules.org](http://www.constraint-handling-rules.org)


- **CHR website** with large bibliography:  