APLAI
Constraint Handling Rules

Gerda Janssens
Departement computerwetenschappen
A01.26
http://www.cs.kuleuven.be/~gerda/APLAI

Overview

1. CHR introduction
2. CHR General Programs
3. CHR Solver Programs
4. Using CHR

Practicalities

Use the CHR library of SWIProlog (not of ECLiPSe!!!!!): chapter 9 of the manual

Book in CBA: Constraint Handling Rules, Thom Fruhwirth, Cambridge University Press, 2009

(Part 1: CHR tutorial)

Practicalities: running experiments

- Example programs
  http://dtai.cs.kuleuven.be/CHR/index.shtml
  http://chr.informatik.uni-ulm.de/~webchr/
- Use constraint declarations with mode and type specifiers! (section 9.3.2 swi manual)
  :- chr_constraint alldifferent(?list(int)).
  :- chr_constraint gcd(+natural).
- Section 9.2.2:
  :- chr_option(debug,off).
  :- chr_option(optimize,full).

A mode is one of:

- **-** The corresponding argument of every occurrence of the constraint is always unbound.
- **+** The corresponding argument of every occurrence of the constraint is always ground.
- **?** The corresponding argument of every occurrence of the constraint can have any instantiation, which may change over time. This is the default value.

1. CHR INTRODUCTION

http://dtai.cs.kuleuven.be/CHR/

- A rule-based programming language, embedded in a host language (Prolog,Java,..)
- Created by Thom Frühwirth in 1991, the CHR language has become a major specification and implementation language for constraint-based algorithms and applications.
- A high-level language for concurrent logical systems.
- A committed-choice language consisting of guarded rules with multiple head atoms.

Industrial users of CHR

Stock Broking - Contact: Mike Elston, Director of Systems Development, SecuritEase, Wellington, New Zealand.

Injection Mold Design - Contact: Alan Baljeu, President, Cornerstone Intelligent Software Corp, Windsor, Canada.

Optical Network Routing - Contact: Jonathan Weston-Dawkes, Mitre Corp, Bedford, USA.

Automatic Test Data Generation - Contact: Ralf Gerlich, BSSE, Immenstaad, Germany.

Unit testing in Java - Agitar Technologies, Cranston, USA.

Programming in CHR

```
main <=> hello_world.
```

Simplification rule used to **rewrite** the constraint `main` (the head) into the constraint `hello_world` (the body)

```
16 ?- main.
hello_world
```

The **constraint store** is a multiset of constraints.

A multiset is a set but elements can occur more than once.

Consider `?- main,main.`

Multi-headed rules

```
orange, gin <=> drink_moete1.
cola, vodka <=> drink_moete2.
drink_moete1, drink_moete2 <=>
                         hangover, blackout.
```

At least one head !!!

Traffic Light Controller

- A traffic light has three states. Its transitions are expressed by the rules:

```
red <=> green.
green <=> orange.
orange <=> red.
```

- Control by a timing device: new `tick` constraint

```
tick, red <=> green.
tick, green <=> orange.
tick, orange <=> red.
```

- Note that a tick constraint fires one rule!!

Rule priorities

```
malfunction, tick <=> malfunction.
tick, red <=> green.
tick, green <=> orange.
tick, orange <=> red.
```

Control by the timer has to be overruled when there is a hardware failure.

Rules are tried in textual order.

```
gentick <=> sleep(7), tick, gentick.
```

Simpagation rule

```
malfunction, tick <=> malfunction.
```

- Delete and add again can be avoided:

```
malfunction \ tick <=> true.
```

`true` represents an empty multiset of constraints

Constraints with arguments

```
:- use_module(library(chr)).     % SWI-prolog!!!
:- chr_constraint light/2 .
:- chr_constraint tick/0, gentick/0,start/0.

light(0, red) <=> write(red),nl,light(60, green).
light(0, green) <=> light(5, orange).
light(0, orange) <=> light(150, red).
tick,light(NumTicks, Color) <=> NTs1 is NumTicks -1,
   light(NTs1,Color).

gentick <=> tick, sleep(1), gentick .
start <=> light(1,red), gentick.

?- start.
red
red
red
```

Propagation rule

- No constraint should be removed
- When two opposing lights are green at the same time

```
light(_, green), light(_,green) ==>
  malfunction.
```

A simple constraint solver

```
:- use_module(library(chr)).
:- chr_constraint(and/3).   % arg 3 is the conjunction
                             % of arg 1 and arg 2
and(0,0,0) <=> true.
and(0,0,1) <=> fail.
and(0,1,0) <=> true.
and(0,1,1) <=> fail.
and(1,0,0) <=> true.
and(1,0,1) <=> fail.
and(1,1,0) <=> fail.
and(1,1,1) <=> true.

?- and(1,1,1)       ?- and(1,0,1).    ?- and(1,1,Z).
Yes                 No                ??
```

A simple constraint solver

```
and(0,0,0) <=> true.
and(0,0,1) <=> fail.
and(0,1,0) <=> true.
and(0,1,1) <=> fail.
and(1,0,0) <=> true.
and(1,0,1) <=> fail.
and(1,1,0) <=> fail.
and(1,1,1) <=> true.

?- and(1,1,Z).
% look for a rule whose lhs is a pattern for and(1,1,Z)
% look for a rule whose lhs is more general than and(1,1,Z)
```

More abstract rules for and/3

```
and(X,X,X) <=> true.

and(0,Y,Z) <=> Z = 0.
% Z has to be 0 for the constraint to be true
% and-constraint can be replaced by the simpler one

and(X,Y,0) <=> not_both(X,Y).   % new chr_constraint
        % not both X and Y are 1

not_both(0,Y) <=> true.
not_both(X,0) <=> true.
not_both(1,Y) <=> Y = 0.
not_both(X,1) <=> X = 0.
```

and/3 version 2

```
and(0,Y,Z) <=> Z = 0.
and(X,0,Z) <=> Z = 0.
and(X,Y,0) <=> not_both(X,Y).
and(1,Y,Z) <=> Z = Y.
and(X,1,Z) <=> Z = X.
and(X,Y,1) <=> X = 1, Y = 1.
and(X,X,Z) <=> X = Z.

24 ?- and(1,1,L).
% look for a rule whose lhs is a pattern for and(1,1,Z)
% look for a rule whose lhs is more general than and(1,1,Z)

L = 1
   % instance of the 4th rule, the 5th rule and the last rule
```

and/3 version 2

```
and(0,Y,Z) <=> Z = 0.
and(X,0,Z) <=> Z = 0.
and(X,Y,0) <=> not_both(X,Y).
and(1,Y,Z) <=> Z = Y.
and(X,1,Z) <=> Z = X.
and(X,Y,1) <=> X = 1, Y = 1.
and(X,X,Z) <=> X = Z.

24 ?- and(X,Y,1).     ?-and(X,Y,0).
    X = 1  Y = 1          not_both(X,Y)
```

% instance of 6th rule; 3rd rule

and/3 version 2 with guards

```
% before: 1st rule was     and(0,Y,Z) <=> Z = 0.
and(X,Y,Z) <=> X == 0 | Z = 0.
and(X,Y,Z) <=> Y == 0 | Z = 0.
and(X,Y,Z) <=> Z == 0 | not_both(X,Y).
and(X,Y,Z) <=> X == 1 | Z = Y.
and(X,Y,Z) <=> Y == 1 | Z = X.
and(X,Y,Z) <=> Z == 1 | X = 1, Y = 1.
and(X,Y,Z) <=> X == Y | X = Z.

24 ?- and(1,1,L). ?- and(X,Y,1).    ?-and(X,Y,0).
L = 1             X = 1  Y = 1        not_both(X,Y)
```

and/3 version 3

```
and(X,0,Z) <=> Z = 0.
and(0,Y,Z) <=> Z = 0.
and(X,Y,0) <=> not_both(X,Y).
and(1,Y,Z) <=> Z = Y.
and(X,1,Z) <=> Z = X.
and(X,Y,1) <=> X = 1, Y = 1.
and(X,X,Z) <=> X = Z.
and(X,Y,Z) \ and(Z,Y,X) <=> Z = X.

?- and(X,Y,Z),and(Z,Y,X).
and(_G133479, _G133480, _G133479)  %  and(X,Y,X)

X = _G133479{user = ...}
Y = _G133480{user = ...}
Z = _G133479{user = ...} ;
```

Another CHR example with guards

```
% http://chr.informatik.uni-ulm.de/~webchr/?load=_min/gcd.pl

% Greatest Common Divisor
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N =< M | L is M-N, gcd(L).

?- gcd(6), gcd(9).
gcd(3).

Yes
```

Two kinds of constraints

- CHR constraints: user-defined
  e.g., `light/2, and/3, gcd/1`
- Built-in constraints: solved by a built-in solver
  - Include syntactic equality `(==/2)`, `=/2, true, fail`
  - Allows to embed existing constraint solvers and side-effect-free host language statements

CHR has 3 types of rules

```
simplification @ H  <=> G | B.
propagation    @ H  ==> G | B.
simpagation    @ H \ H'<=> G | B.
```

Name of the rule (to the left of @) is optional: unique identifier of the rule

The head (H, H') is a non-empty conjunction of CHR constraints.

The guard G is a conjunction of built-in constraints.

The body B is a goal, namely a conjunction of built-in and CHR constraints.

Simpagation is abbreviation for: H ^ H' <=> C | H ^ B.

Logical reading (semantics)

```
H  <=> G | B.
```
$\forall X(G \rightarrow (H \leftrightarrow \exists Y\ B))$

```
H  ==> G | B.
```
$\forall X(G \rightarrow (H \rightarrow \exists Y\ B))$

```
max(X,Y,Z) <=> X =< Y | Z = Y .
```

$\forall X,Y,Z\ (X<=Y \rightarrow (max(X,Y,Z) \leftrightarrow Z = Y))$

Operational semantics

- CHR program : provide an initial state and apply rules until no more rules are applicable (fixpoint is reached) or a contradiction occurs
- Transition from one state to another
- States are goals, namely conjunctions of built-in constraints and CHR constraints
- Other name for state: constraint store

Rule application

```
simplification @ H  <=> G | B.
```

- Replaces instances of the CHR constraint H in the constraint store by B on condition that the guard G holds

```
propagation    @ H  ==> G | B.
```

- Similar, but now H is kept and B is added to the store.

When is a rule applicable?

- If all the head constraints of the rule are matched by constraints in the current goal one-by-one;
- And if under this matching the guard of the rule holds
- Any of the applicable rules can be applied, and the application cannot be undone (committed choice).

If all its head constraints are matched by constraints

- Similar to matching in Haskell
- The store should contain a constraint hs such that hs is an instance of the head constraint hr
- Thus only 1 directional unification: $hs = (hr)\ \theta$
- The variables in the query constraints are not bound by the matching

Meaning of matching

- `c(X) <=> true.`
- `c(a) <=> true.`
- `c(f(A)) <=> true.`
- `c(X,X) <=> true.`
- `c(X), d(X) <=> true.`
- `c(X) <=> true | true.`
- `c(X) <=> X == a | true.`
- `c(X) <=> nonvar(X), X = f(A) |true.`
- `c(X,Y) <=> X == Y | true.`
- `c(X), d(Y) <=> X == Y | true.`

Rule application

- If new constraints are added to the constraint store, rule applications are restarted.
- Computation stops in a failed final state if the built-in constraints become inconsistent.
- Trivial non-termination of the propagation step is avoided by applying a propagation rule at most once to the same constraints.

Implementation of CHR: refined semantics

- Treats constraints from left to right and as procedure calls.
- Current constraint = active constraint
- Look for possible applicable rules in order, until all matching rules have been executed or the constraint is deleted from the store.
- When a matching rule fires, the constraints in the body are executed
- When they finish, the execution returns to finding rules for the current active constraint.

Example

```
1 @ gcd(0) <=> true.
2 @ gcd(N) \ gcd(M) <=> N =< M | L is M-N, gcd(L).
```

GOAL STORE

```
{gcd(6),gcd(9)}  ∅
{gcd(9)}  {gcd(6)}
∅  {gcd(6), gcd(9)}
Rule 2: N = 6 and M = 9
∅  {gcd(6), gcd(3)}
Rule 2: N = 3 and M = 6
∅  {gcd(3), gcd(3)}
Rule 2: N = 3 and M = 3
∅  {gcd(3), gcd(0)}
Rule 1
∅  {gcd(3)}    % final state as no rules are applicable
```

In detail ….

```
8 ?- gcd(6),gcd(9).
CHR:   (0) Insert: gcd(6) # <187>
CHR:   (1) Call: gcd(6) # <187> ? [creep]
CHR:   (1) Exit: gcd(6) # <187> ? [creep]
CHR:   (0) Insert: gcd(9) # <188>
CHR:   (1) Call: gcd(9) # <188> ? [creep]
CHR:   (1) Try: gcd(6) # <187> \ gcd(9) # <188> <=>
  6=<9 | _G24620 is 9-6, gcd(_G24620).
CHR:   (1) Apply: gcd(6) # <187> \ gcd(9) # <188>
  <=> 6=<9 | _G24620 is 9-6, gcd(_G24620). ? [creep]
CHR:   (1) Remove: gcd(9) # <188>
CHR:   (1) Insert: gcd(3) # <189>
CHR:   (2) Call: gcd(3) # <189> ? [creep]
```

```
CHR:   (2) Call: gcd(3) # <189> ? [creep]
CHR:   (2) Try: gcd(3) # <189> \ gcd(6) # <187> <=>
   3=<_G25307 | _G25312 is _G25307-3, gcd(_G25312).
CHR:   (2) Apply: gcd(3) # <189> \ gcd(6) # <187>
   <=> 3=<_G25307 | _G25312 is _G25307-3,
   gcd(_G25312). ? [creep]
CHR:   (2) Remove: gcd(6) # <187>
CHR:   (2) Insert: gcd(3) # <190>
CHR:   (3) Call: gcd(3) # <190> ? [creep]
CHR:   (3) Try: gcd(3) # <189> \ gcd(3) # <190> <=>
   3=<3 | _G26008 is 3-3, gcd(_G26008).
CHR:   (3) Apply: gcd(3) # <189> \ gcd(3) # <190>
   <=> 3=<3 | _G26008 is 3-3, gcd(_G26008). ? [creep]
CHR:   (3) Remove: gcd(3) # <190>
CHR:   (3) Insert: gcd(0) # <191>
CHR:   (4) Call: gcd(0) # <191> ? [creep]
```

```
CHR:   (3) Insert: gcd(0) # <191>
CHR:   (4) Call: gcd(0) # <191> ? [creep]
CHR:   (4) Try: gcd(0) # <191> <=> true.
CHR:   (4) Apply: gcd(0) # <191> <=> true. ? [creep]
CHR:   (4) Remove: gcd(0) # <191>
CHR:   (4) Exit: gcd(0) # <191> ? [creep]
CHR:   (3) Exit: gcd(3) # <190> ? [creep]
CHR:   (2) Exit: gcd(3) # <189> ? [creep]
CHR:   (1) Exit: gcd(9) # <188> ? [creep]
gcd(3)
true ;
CHR:   (1) Redo: gcd(9) # <188>
….
CHR:   (0) Fail: gcd(6) # <187> ? [creep]
false.
```

Debugging (1)

```
7 ?- chr_trace.
Yes
8 ?- gcd(45), gcd(25).
CHR:   (0) Insert: gcd(45) # <173>
CHR:   (1) Call: gcd(45) # <173> ? [creep]
CHR:   (1) Exit: gcd(45) # <173> ? [creep]
CHR:   (0) Insert: gcd(25) # <174>
CHR:   (1) Call: gcd(25) # <174> ? [creep]
CHR:   (1) Try: gcd(25) # <174> \ gcd(45) # <173> <=>
   25=<_G23943 | _G23948 is _G23943-25, gcd(_G23948).
CHR:   (1) Apply: gcd(25) # <174> \ gcd(45) # <173> <=>
   25=<_G23943 | _G23948 is _G23943-25, gcd(_G23948). ?
   [break]
```

Debugging (2)

```
CHR:   (1) Apply: gcd(25) # <174> \ gcd(45) # <173> <=>
   25=<_G23943 | _G23948 is _G23943-25, gcd(_G23948). ?
   [break]
% Break level 1
[1] 9 ?- chr_show_store(user).
gcd(25)
gcd(45)
gcd(25)
gcd(45)
```

Artificial Example: simplification

```
gg(X) <=> X = a, write(rule1).
gg(X) <=> X = b, write(rule2).

13 ?- gg(X) .
% the first rule is applied thus
rule1
X = a ;
% what about the second rule???
% no more gg/1 constraint
% committed choice for simplifications
No
```

AE cont. propagation

```
gg(X) ==> X = a, write(rule1).
gg(X) ==>  write(rule2).

11 ?- gg(X).
rule1
% what with rule2??
% also fires !!!
rule2
X = a ;
No
```

AE cont. propagation

```
gg(X) ==> X = a, write(rule1).
gg(X) ==> X = b, write(rule2).

13 ?- gg(X).
Rule1
% what about rule2??
No
```

AE cont. Role of the ***guards***

```
gg(X) <=> X == a| write(rule1).
gg(X) <=> X == b| write(rule2).

5 ?- gg(X).
X = _G144999{user = ...}
gg(_G144999)

Yes
7 ?- gg(b).
rule2

Yes
9 ?- gg(X), X = b.
rule2

X = b ;

No
```

```
17 ?- gg(X), X = b .
CHR:   (0) Insert: gg(_G145337) # <574>
CHR:   (1) Call: gg(_G145337) # <574> ? [creep]
CHR:   (1) Exit: gg(_G145337) # <574> ? [creep]
CHR:   (1) Wake: gg(b) # <574> ? [creep]
CHR:   (1) Try: gg(b) # <574> <=> b=b | write(rule2).
CHR:   (1) Apply: gg(b) # <574> <=> b=b | write(rule2). ?
   [creep]
CHR:   (1) Remove: gg(b) # <574>rule2
CHR:   (1) Exit: gg(b) # <574> ? [creep]

X = b ;
CHR:   (1) Redo: gg(b) # <574>
CHR:   (0) Fail: gg(b) # <574> ? [creep]
CHR:   (1) Redo: gg(_G145337) # <574>
CHR:   (0) Fail: gg(_G145337) # <574> ? [creep]

No
```

Reactivation example

```
% not equal
neq(X,X) <=> fail.
neq(X,Y) <=> X \= 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.
No
?- neq(A,B), A = a, B = b.
Yes
?- neq(A,B), A = f(C), B = f(D).
neq(f(C),f(D))
```

Prolog vs CHR

- Heads: 1 vs at least one
- rule selection: unication vs matching & guard
- different rules: alternatives/backtracking vs try all in sequence
- no rule: failure vs delay

2. CHR GENERAL PROGRAMS

CHR program: prime numbers

```
:- use_module(library(chr)).   % SWI
:- chr_constraint candidate/1, prime/1 .

candidate(1) <=> true.
candidate(N) <=>
      N > 1 | prime(N), N1 is N-1, candidate(N1).

absorb(X) @ prime(Y) \ prime(X) <=> 0 is X mod Y | true .

?- candidate(8).
           % when does the absorb rule fires for the 1st time
prime(2)
prime(3)
prime(5)
prime(7)
```

CHR program: path in a graph

```
:- use_module(library(chr)).
:- chr_constraint edge/2, path/2 .

rem_dupl @ path(X,Y) \ path(X,Y) <=> true.       % a set!!!
path_1   @ edge(X,Y) ==> path(X,Y)
path_add @ path(X,Y), path(Y,Z) ==>
                X \== Y, Y \== Z  | path(X,Z).

6 ?- edge(1,2), edge(2,3),edge(2,4).
edge(2, 4)  edge(2, 3) edge(1, 2)
path(1, 4)  path(2, 4) path(1, 3)  path(2, 3)  path(1, 2)
```

In fact, we compute the transitive closure of the binary relation given by edge(X,Y), namely the (X,Y) in path(X,Y)

CHR representation of a set of elements

- As a Prolog list ???

- Or better: as a set of constraints!!!
- E.g. A set containing the elements 1, 5 and 3
  `set_item(s1,1). set_item(s1,5). set_item(s1,3).`
- Better for CHR-style programming!!!

CHR program: sorted list for a set of natural numbers

- Representation of the set: item/1
- Representation of the sorted list: link/2

```
start   @ item(X) <=> link(0, X).
trivial @ link(X,X) <=> true.
sort  @ link(X,Y) \ link(X,Z) <=>
            X < Y, Y =< Z | link(Y,Z).

14 ?- item(1),item(5),item(3).
link(3,5)
link(1,3)
link(0,1)
Yes
```

CHR program: map colouring

```
:- chr_constraint n/2, hascolor/2.  %neigb/2 facts as inputdata

setsem  @ hascolor(X,Cx) \ hascolor(X,Cy) <=> Cx = Cy.
diffcol @ n(X,Y),hascolor(X,Cx), hascolor(Y,Cy) ==> Cx \== Cy.

main :- findall(n(X,Y), neigb(X,Y), L), setconstraints(L).

setconstraints([]).
setconstraints([n(X,Y) | T] ) :-
   n(X,Y), setcolor(X), setcolor(Y), setconstraints(T).

setcolor(X) :- color(C), hascolor(X,C).     % search by Prolog
                                            % constraints ??
?- main.
hascolor(br, green)       n(top,bl) ….
hascolor(bm, blue)
hascolor(bl, green)
hascolor(top, red)
```

Programming Notes (1)

- Enforcing a symmetric relation

```
symmrel(X,Y) \ symmrel(X,Y) <=> true.
symmrel(X,Y) ==> symmrel(Y,X).
```

- Or keep only one version of the constraint

```
symmrel(X,Y) \ symmrel(X,Y) <=> true.
symmrel(X,Y) <=> Y @< X | symmrel(Y,X).
```

- Representation of data as constraints

```
element(R1,K), element(R2,K)
```

PN(2): CHR and backtracking

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

?- p.
c1 ;
d1
```

- Prolog creates choicepoints
- CHR does not
- Prolog backtracking undoes CHR changes

PN(3): Control of execution by Phases

- Divide the program into different phases (why?)
- Rules are active: waiting for matching constraints
- Sometimes:
  - first all data as constraints in store;
  - then we start with pruning.

PN(3): Control of execution by Phases

- Divide the program into different phases (why?)

```
% phase 1
r1 @ phase1 \ pred1(arg1,arg2) <=>
               guard | pred1(arg1,arg3).
% rule r1 fires as long as there are
  constraints in the store satisfying the head
  and the guard
tophase2 @ phase1 <=> phase2.
% phase 2
r2 @ phase2 \ pred(…,…) <=>
```

PN(4): Debugging

- `?- chr_trace.`
- To enumerate all foo/2 constraints (not to be used in a CHR program as it is **bad programming practice**; thus for debugging only!!!)

```
(find_chr_constraint(foo(X,Y)),
     writeln(foo(X,Y)),
     fail
    ;
     true
   )
```

Some exercises

- Compute the Nth Fibonacci number:
- fib(N,M) is true if M is the Nth Fibonacci number.
  - Compute them top-down: goal driven, backward-chaining
  - Compute them bottom-up: data driven; forward-chaining

CHR SOLVER PROGRAMS

Overview CHR constraint solvers

- The example solver: leq/2
- Finite domain solver
- SEND MORE MONEY and N-queens
- (Linear equations)

The leq/2 solver (less or equal)

```
:- chr_constraint leq/2.
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).


1 ?-  leq(A,B), leq(C,A), leq(B,C).     % A = C and B = C

% constraint store
leq(A,B).
leq(A,B), leq(C,A)      % transitivity  for leq(C,A)
leq(A,B), leq(C,A), leq(C,B)  % no rule fires for leq(C,B)
leq(A,B), leq(C,A), leq(C,B), leq(B,C)      % antisymm
leq(A,B), leq(C,A), B = C
                                     % wake leq(B,A) antisymm
A = B, B = C
```

The leq/2 solver

```
:- chr_constraint leq/2.
reflexivity  @ leq(X,X) <=> true.
antisymmetry @ leq(X,Y), leq(Y,X) <=> X = Y.
transitivity @ leq(X,Y), leq(Y,Z) ==> leq(X,Z).


2 ?- leq(A,B), leq(B,C).
leq(_G38414, _G38418)        % leq(A,C)
leq(_G38415, _G38418)
leq(_G38414, _G38415)

A = _G38414{leq = ...},
B = _G38415{leq = ...},
C = _G38418{leq = ...}
```

CHR finite domain solver

- The domain constraint X in D means that the variable X takes its value from the given finite domain D.
- Bounds consistency for interval constraints
- Implementation based on interval arithmetic
- Built-in (i.e. Prolog) constraints <, >, =<, >=, \==, min, max, +, -

- CHR: X in A..B constrains X to be in the interval A..B

```
:-op( 700,xfx,in).
:-op( 700,xfx,le).
:-op( 700,xfx,eq).
:-op( 600,xfx,'..').

:- chr_constraint le/2, eq/2, in/2, add/3.
```

Finite domain solver in/2 le/2

```
inconsistency @ X in A..B <=> A > B | fail.
intersect     @ X in A..B, X in C..D <=>
                     X in max(A,C)..min(B,D) .

le @ X le Y, Y in C..D \ X in A..B <=> B > D | X in A..D.
le @ X le Y, X in A..B \ Y in C..D <=> C < A | Y in A..D.

16 ?- U in 2..3, V in 1..2 , U le V .
_G60176 le _G60182  _G60182 in 2..2   _G60176 in 2..2

U = _G60176{user = ...},
V = _G60182{user = ...} ;

No
17 ?- U in 5..6, V in 1..2, U le V .
No
```

Finite domain solver eq/2 add/3

```
eq @ X eq Y \ X in A..B, Y in C..D <=> A \== C |
         L is max(A,C), X in L..D, Y in L..D.
eq @ X eq Y \ X in A..B, Y in C..D <=> B \== D |
         U is min(B,D), X in A..U, Y in A..U.

% guard: at least one interval is reduced !!
add @ add(X,Y,Z) \ X in A..B, Y in C..D, Z in E..F  <=>
   not(( A >= E-D, B =< F -C, C >= E-B, D =< F-A, E >= A+C,
   F =< B+D)) |
   Lx is max(A,E-D), Ux is min(B,F-C), X in Lx..Ux,
   Ly is max(C,E-B), Uy is min(D,F-A), Y in Ly..Uy,
   Lz is max(E,A+C), Uz is min(F,B+D), Z in Lz..Uz.

?- U in 1..3, V in 2..4, W in 0..4, add(U,V,W).
W in 3..4   V in 2..3  U in 1..2
add(U,V,W)
```

FD solver: enumeration domains

```
% explicit enumeration of values: [red,blue,green]
inconsistency @ X in []  <=> fail.
intersect     @ X in L1, X in L2  <=> intersect(L1,L2,L3) |
  X in L3.
% similar to bounds consistency for le/2 ….

% search      or in Prolog??
enum([]) <=> true.
enum([X|L]) <=> indomain(X), enum(L).

indomain(X), X in [V|L] <=> L = [_|_] |
  (X in [V] ; X in L, indomain(X)).

indomain(X), X in A..B <=> A < B |
  Middle is (A+B) // 2,
  ( X in A..Middle, indomain(X)
  ;
     M1 is Middle+1,  X in M1 .. B, indomain(X)).
```

Use FD solver to solve n-queens

- Domain constraints??

- Queens constraints??

  once the row of a queen is fixed (X in [V] !!),

  forward checking

  the other queens should be “safe”

  at distance N: V-N and V+N

N queens

```
solve(N,Qs) :- makedomains(N,Qs) , queens(Qs)  , enum(Qs).

queens([Q|Qs]) <=> safe(Q,Qs,1), queens(Qs).

safe(X,[Y|Ys],N) <=> noattack(X,Y,N), N1 is N + 1, safe(X,Ys,N1).

% noattack(X,Y,N): queen X doesn't attack queen Y (column distance = N)
noattack(X,Y,N), X in [V] \ Y in L  <=>  X1 is V-N, X2 is V+N,
        delete(L,V,L1), delete(L1,X1,L2), delete(L2,X2,L0), L \== L0 |
   Y in L0.
noattack(Y,X,N), X in [V] \  Y in L  <=>  X1 is V-N, X2 is V+N,
        delete(L,V,L1), delete(L1,X1,L2), delete(L2,X2,L0), L \== L0 |
   Y in L0.

% Can delete fail?
% delete(+List1, ?Elem, ?List2)
%    Delete all members of List1 that simultaneously unify with Elem %
%    and unify the result with List2.
% Why L\== L0?
```

N queens

```
makedomains(N,Qs) :- genlist(1,N,Doms), makedomains(N,Qs,Doms).
makedomains(0,[],_).
makedomains(M,[Q|Qs],Doms) :- M > 0,
   Q in Doms, M1 is M -1, makedomains(M1,Qs,Doms).


genlist(N,N,[N]).
genlist(M,N,[M|T]) :- M< N, M1 is M + 1, genlist(M1,N,T).
```

Execution ?- solve(4,K).

```
_G126568 in[3]
_G126493 in[1]
_G126418 in[4]
_G126342 in[2]
indomain(_G126568)                %from finite domain solver enumeration
indomain(_G126493)
indomain(_G126418)
queens([])
safe(_G126568, [], 1)
safe(_G126493, [], 2)
safe(_G126418, [], 3)
safe(_G126342, [], 4)
noattack(_G126493, _G126568, 1)
noattack(_G126418, _G126568, 2)
noattack(_G126418, _G126493, 1)
noattack(_G126342, _G126568, 3)
noattack(_G126342, _G126493, 2)
noattack(_G126342, _G126418, 1)

K = [_G126342{user = ...}, _G126418{user = ...}, _G126493{user = ...},
   _G126568{user = ...}]
```

When noattack fires… q1 on 1

```
noattack(X,Y,N), X in [V] \ Y in L  <=>  X1 is V-N, X2 is V+N,
        delete(L,V,L1), delete(L1,X1,L2), delete(L2,X2,L0), L \== L0 |
   write(now1(Y,L,L0)), Y in L0.

now1(_G126568, [1, 2, 3, 4], [2, 3])        % q4  V(1)  N(3)
now1(_G126493, [1, 2, 3, 4], [2, 4])        % q3
now1(_G126418, [1, 2, 3, 4], [3, 4])        % q2
now1(_G126493, [2, 4], [])                  % q2 on 3, thus
backtrackingon(_G126418)
now1(_G126493, [2, 4], [2])                 % q2 on 4, thus
now1(_G126568, [2, 3], [])
backtrackingon(_G126342)                    % q1 on 2
now1(_G126568, [1, 2, 3, 4], [1, 3, 4])
now1(_G126493, [1, 2, 3, 4], [1, 3])
now1(_G126418, [1, 2, 3, 4], [4])
now1(_G126493, [1, 3], [1])
now1(_G126568, [1, 3, 4], [3, 4])
now1(_G126568, [3, 4], [3])

qq([_G126342, _G126418, _G126493, _G126568])
```

SEND + MORE = MONEY

- Also as FD problem
- Domain constraints: S in 0..9
- Sum?   A + B = C??

equation(A+B,C) <=> ???? | A+B =:= C.

- All different?? diff/2

diff(X,Y) with fixed values for X and Y, then …

diff(X,Y) with fixed value of Y, then ….

SEND + MORE = MONEY

```
:- chr_constraint indomain/1, in/2, diff/2, enum/1, equation/1 .
:-op( 700,xfx,in).

send(Sol) :- Sol = [S,E,N,D,M,O,R,Y],
   make_domain(Sol,[0,1,2,3,4,5,6,7,8,9]),
   diff(S,0), diff(M,0),
   equation(Sol), enum(Sol).
% make_domain(L,D) enforces for each element E of L:
% E is from domain D and different to each other element
   from L
make_domain([],_).
make_domain([X|L],D) :- all_different(L,X), X in D,
   make_domain(L,D).
% all_different(L,E) enforces that each element of L is
   different to E
all_different([],_).
all_different([H|T],E) :-
    diff(H,E),
    all_different(T,E).
```

```
inconsistency @ X in []  <=> fail.
singlevalue   @ X in [V] <=> X = V .
intersect     @ X in L1, X in L2  <=> intersect(L1,L2,L3) | X
   in L3.

diff(X,Y) <=> nonvar(X), nonvar(Y) | X \== Y.
diff(Y,X) \ X in L <=> nonvar(Y), select(Y,L,NL) | X in NL.
diff(X,Y) \ X in L  <=> nonvar(Y), select(Y,L,NL) | X in NL.

equation([S,E,N,D,M,O,R,Y])<=> ground([S,E,N,D,M,O,R,Y])|
1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
    =:=  10000*M + 1000*O + 100*N + 10*E + Y.
```

```
enum([]).
enum([X|L]) :- indomain(X), enum(L).

indomain(X), X in [V|L] <=> L = [_|_] |
   ( X in [V]
   ;
     X in L, indomain(X)
   ).
indomain(X) <=> ground(X) | true.
?- time(send(K)).
% 1,880,358,351 inferences, 490.32 CPU in 492.45 seconds
   (100% CPU, 3834962 Lips)    K = [9, 5, 6, 7, 1, 0, 8, 2] ;

with options...
?- time(send(K)).
% 921,528,259 inferences, 256.62 CPU in 257.46 seconds (100%
   CPU, 3591023 Lips)           K = [9, 5, 6, 7, 1, 0, 8, 2] ;
```

SEND + MORE = MONEY

- `:- chr_option(debug,off).`
- `:- chr_option(optimize,full).`
- Why not performant??
- Due to weaker version of alldifferent
- also no linear equation solver

- In general: modes and types (see manual SWI Prolog section 7)
- `:- chr_constraint get_value(+,?).`
- `:- chr_constraint domain(?int,+list(int)), alldifferent(?list(int)).`

4. USING CHR

Using CHR

Applications

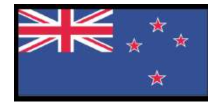
- Your own solver
- Mimick business rules

Applications

- Typically you want more than is provided by the built-in solver: Prolog, ECLiPSe, …
- You can use CHR to transform your constraints into "known constraints".
- You can use CHR to express your own constraint solver
- Companies use CHR for
  - Multi-headed business rules
  - Custom constraint solvers

CHR in industry

- Scientic 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

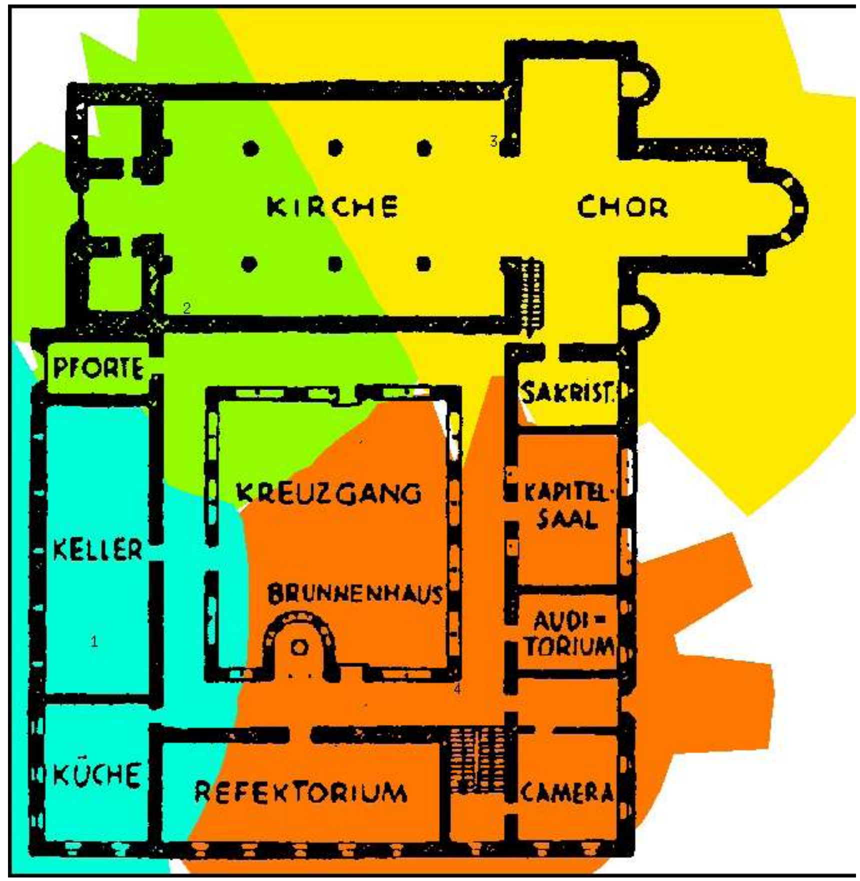
More Typical Application Topics of Constraint Handling Rules

- Abduction
- Agents
- Grammars
- Implementation and Compilation
- Scheduling
- Spatial Reasoning
- Temporal Reasoning
- Software Testing
- Type Systems
- Verification

An example of a solver…

T. Frühwirth, P. Brisset,
Optimal Placement of Base Stations in Wireless Indoor Communication Networks, IEEE Intelligent Systems Magazine 15(1), 2000.

Voted Among Most Innovative Telecom Applications of the Year by IEEE Expert Magazine, Winner of CP98 Telecom Application Award.



Problem description

- To compute the minimal number of base stations and their locations.
- Number of test points representing a possible receiver position.
- For each test point: a radio-cell which is the area in which a sender can be reaching the test point.
- There must be a sender in each radio-cell; covering as many radio-cells at once
- Shape of radio-cell: odd-shaped object
- First approximated in 2D by a rectangle.
- Then refinements …

Constraints on the position of the senders

- A rectangle can be represented by its left lower corner L and its right upper corner R.
- A 2D coordinate: X#Y
- A constraint inside(Sender, L-R) expresses that Sender must be inside the rectangle L-R.

```
non-empty @ inside(S, A#B - C#D) ==> A < C, B < D.
intersect @ inside(S, A1#B1 - C1#D1),
            inside(S, A2#B2 - C2#D2)  <=>
                 A is max(A1,A2), B is max(B1,B2),
                 C is min(C1,C2), D is min(D1,D2),
                 inside(S, A#B - C#D).
```

Labeling phase

```
equate_senders([]) <=> true.
equate_senders([S|L]) <=>
      (member(S,L)     % equate S with another sender
      ;
      true              % or not
      ),
      equate_senders(L).
```

```
% labeling constrains the locations of the senders more
  and more
% heuristic: equate senders that are "neighbors"
```

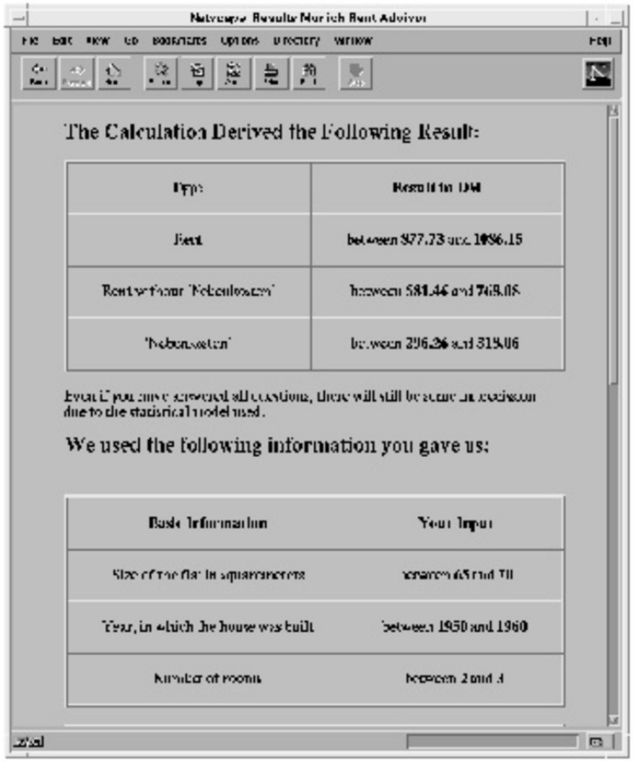
Refinements

```
% extended to work with union of rectangles…

inside(S, [R1,R2, …, Rn])  % in R1 or in R2 or  …


% to deal with 3D
% add restrictions that the senders should be easily
  reachable, e.g. near to walls, …,

% elegant solver: not provided as a black box by
  standard systems
```

The Munich rent advisor

T. Frühwirth,S. Abdennadher
The Munich Rent Advisor,
Journal of Theory and
Practice of Logic
Programming, 2000.

Most Popular
Constraint-Based Internet
Application.
Thom Frühwirth Constraint Handling Rules
(CHR)



Many more

- Spatial-temporal reasoning
- Agents and actions
  - FLUX: A Logic Programming Method for Reasoning Agents
- Types and security
  - Constraint-Based Polymorphic Type Inference
- Testing and verification
  - Model Based Testing for Real:The Inhouse Card Case Study

Link with business rules

- Next topic: rule-based systems such as Jess, JBoss Rules, ….,   BUSINESS RULES
- A famous BR case study : EU car rental application

- **Rental reservation acceptance**
  - If a rental request does not specify a particular car group or model, the default is group A (the lowest-cost group).
  - Reservations may be accepted only up to the capacity of the pick-up branch on the pick-up day.
  - If the customer requesting the rental has been blacklisted, the rental must be refused.
  - A customer may have multiple future reservations, but may have only one car at any time.

Also with CHR rules

```
day(Today) \
   reservation(Renter,
             CarGroup, PickUpDay, PickUpBranch,
             ReturnDay, ReturnBranch),
   isAvailable(car(N,M,CarGroup,M0,X0),
             AvailableDay,PickUpBranch),
   customer(Renter,nonRenting,BadExp)
<=>
    PickUpDay >= AvailableDay, BadExp =< 3
   | rentalagreement(Renter, car(N,M,CarGroup,M0,X0),
       PickUpDay,PickUpBranch,ReturnDay,ReturnBranch),
     customer(Renter,renting,BadExp),
    % do sth with isAvailable constraint…
```

Main difference with Business Rules

- Commercial software
- Interface software, visualisation
  - such as decision tables
- The way rules are activated
  - CHR: uses lazy matching
  - The active constraint must find its partners…

- What could be an alternative???