A tutorial on Constraint Handling Rules (Part 1)

Rémy Haemmerlé
CLIP Group, Universidad Politéctnica de Madrid, Spain

April 25, 2013

What is Constraint Handling Rules (CHR) ?

- A descendant of Constraint Logic Programming (CLP).

What is Constraint Handling Rules (CHR) ?

- A descendant of Constraint Logic Programming (CLP).
- A declarative programming language.

What is Constraint Handling Rules (CHR) ?

- A descendant of Constraint Logic Programming (CLP).
- A declarative programming language.
- A language extension for a host language (e.g. Prolog).

What is Constraint Handling Rules (CHR) ?

- A descendant of Constraint Logic Programming (CLP).
- A declarative programming language.
- A language extension for a host language (e.g. Prolog).
- A decent production rule system.

What is Constraint Handling Rules (CHR) ?

- A descendant of Constraint Logic Programming (CLP).
- A declarative programming language.
- A language extension for a host language (e.g. Prolog).
- A decent production rule system.
- A multiset rewriting formalism with constraints.

What is Constraint Handling Rules (CHR) ?

- A descendant of Constraint Logic Programming (CLP).
- A declarative programming language.
- A language extension for a host language (e.g. Prolog).
- A decent production rule system.
- A multiset rewriting formalism with constraints.

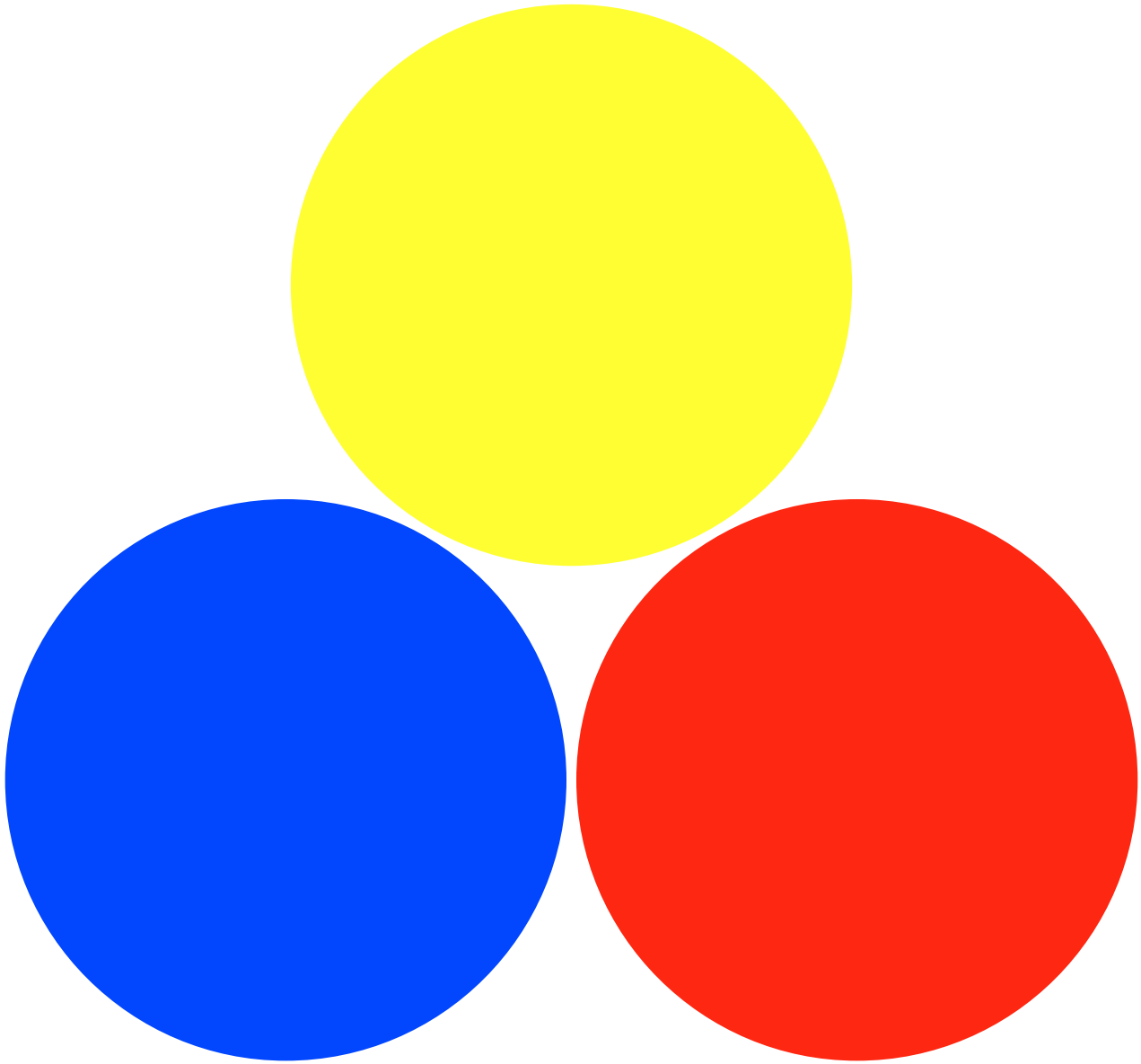
☞ Presentation material (mostly) by :

- Thom Früwirth, Ulm University
- Tom Schrijvers, Gent University
- Jon Sneyers, Leuven University

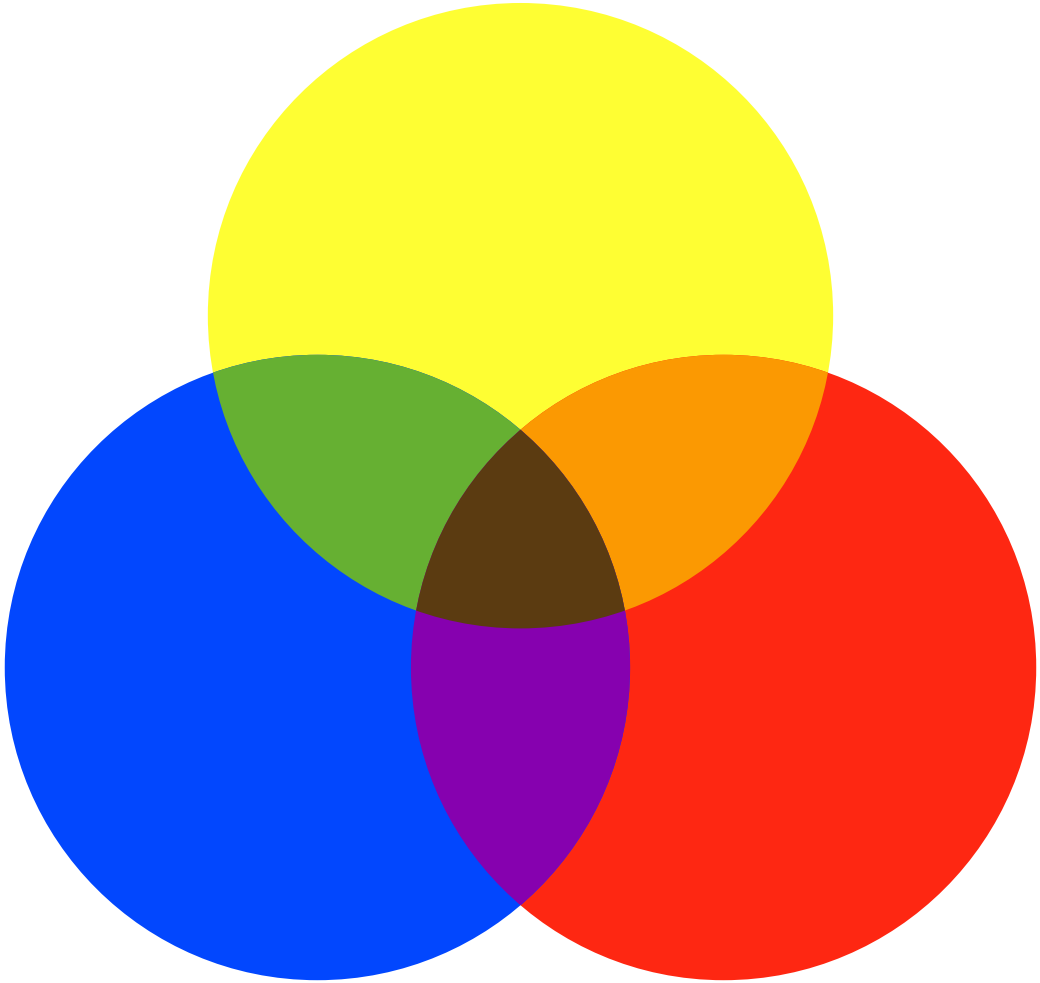
`http://dtai.cs.kuleuven.be/CHR/tutorials.shtml`

Outline

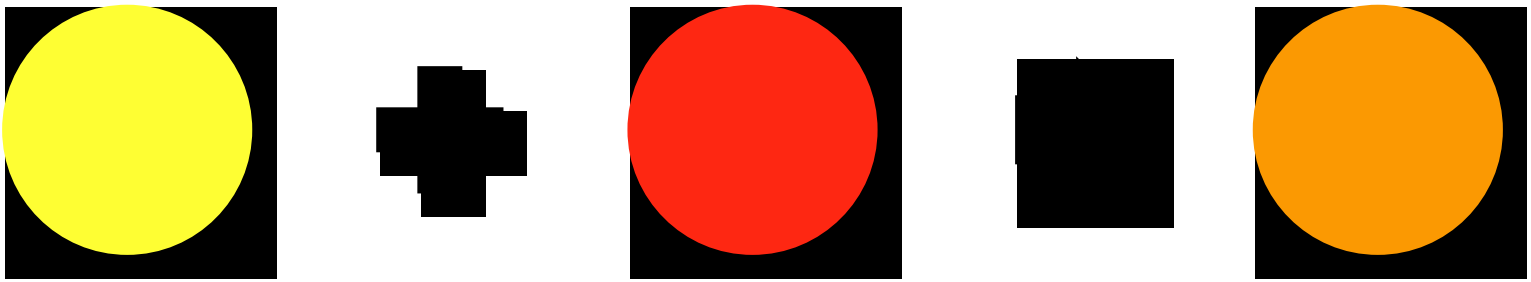
Color Mixing



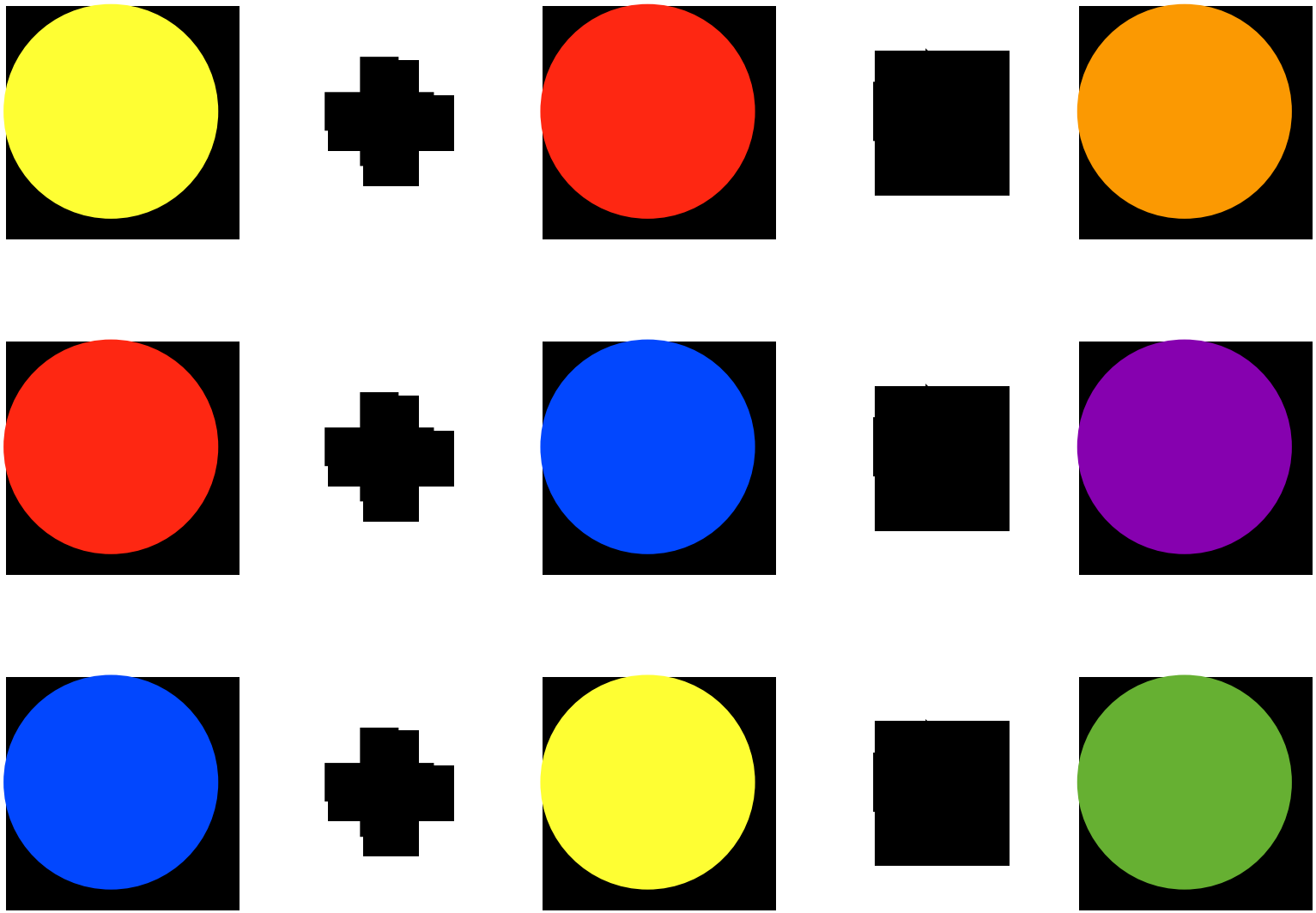
Color Mixing



Mixing colors: the rules



Mixing colors: the rules



Mixing color in CHR

mixing color	**CHR**
paints	(user) constraints
rules	simplification rules
palette	constraints store

Mixing color: paints as constraints

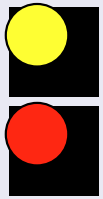
Declaring paints in textual form:

```
:- chr_constraint yellow.                    % 
```

Mixing color: paints as constraints

Declaring paints in textual form:

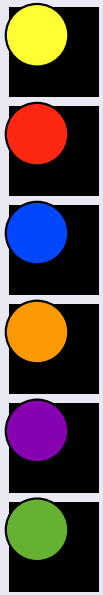
```
:- chr_constraint yellow.                    %
:- chr_constraint red.                       %
```



Mixing color: paints as constraints

Declaring paints in textual form:

```
:- chr_constraint yellow.                    %
:- chr_constraint red.                       %
:- chr_constraint blue.                      %
:- chr_constraint orange.                    %
:- chr_constraint purple.                    %
:- chr_constraint green.                     %
```



Mixing color: paint rules as simplification rules

Simplification rules : `head <=> body.`

Mixing color: paint rules as simplification rules

Simplification rules : `head <=> body.`

"`head` can be replaced by `body`"

Mixing color: paint rules as simplification rules

Simplification rules : `head <=> body.`

"`head` can be replaced by `body`"

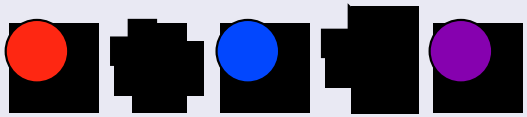
```
% 
yellow, red <=> orange
```

Mixing color: paint rules as simplification rules

Simplification rules : `head <=> body.`

"`head` can be replaced by `body`"

```
%
```


```
yellow, red <=> orange
%
```


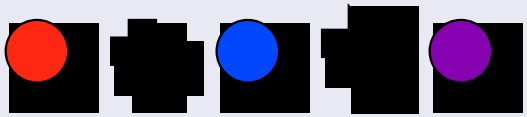
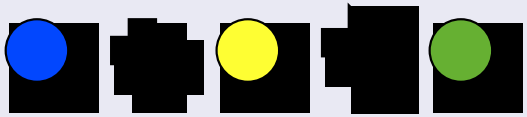
```
red, blue <=> purple
```

Mixing color: paint rules as simplification rules

Simplification rules : `head <=> body.`

"`head` can be replaced by `body`"

```
%
```

```
yellow, red <=> orange
%
```

```
red, blue <=> purple
%
```

```
blue, yellow <=> green
```

Mixing color effectively

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

CHR execution:

- Rules are applied exhaustively to the user **query**.
- The remaining constraints form the **result**.

Mixing color effectively

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

CHR execution:

- Rules are applied exhaustively to the user **query**.
- The remaining constraints form the **result**.

```
?- yellow, red
orange
```

Mixing color effectively

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

CHR execution:

- Rules are applied exhaustively to the user **query**.
- The remaining constraints form the **result**.

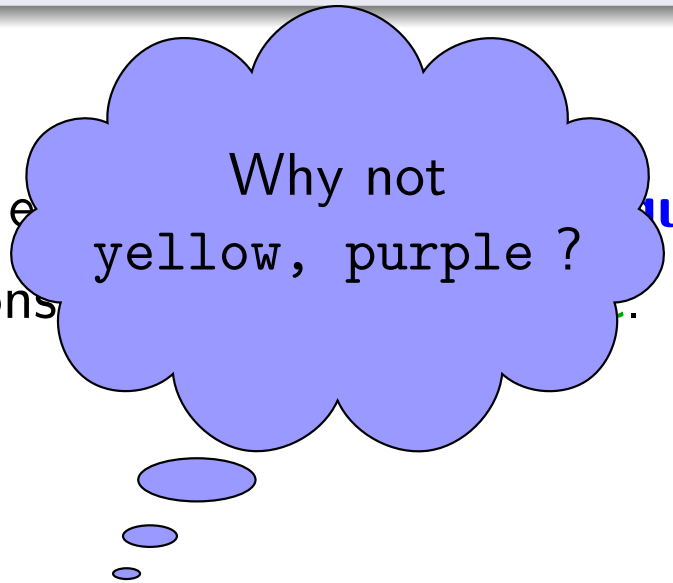
```
?- yellow, red
orange
?- yellow, red, blue
orange,
blue
```

Mixing color effectively

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

CHR execution:

- Rules are applied e… …uery.
- The remaining cons… …




```
?- yellow, red
orange
?- yellow, red, blue
orange,
blue
```

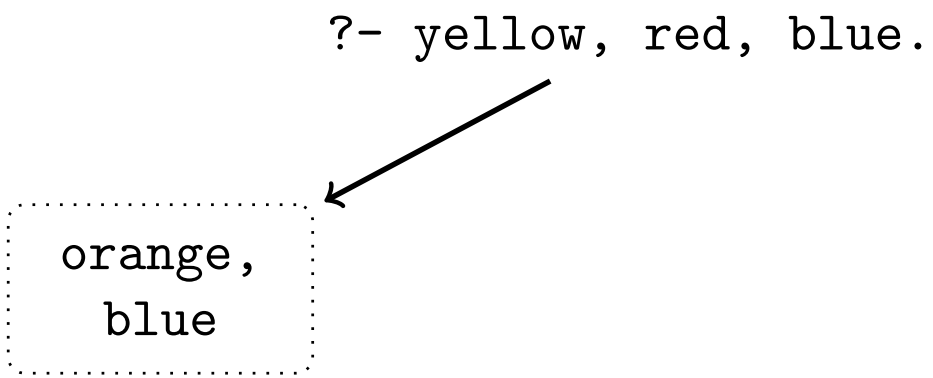
Abstract semantics versus refined Semantics

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

```
?- yellow, red, blue.
```

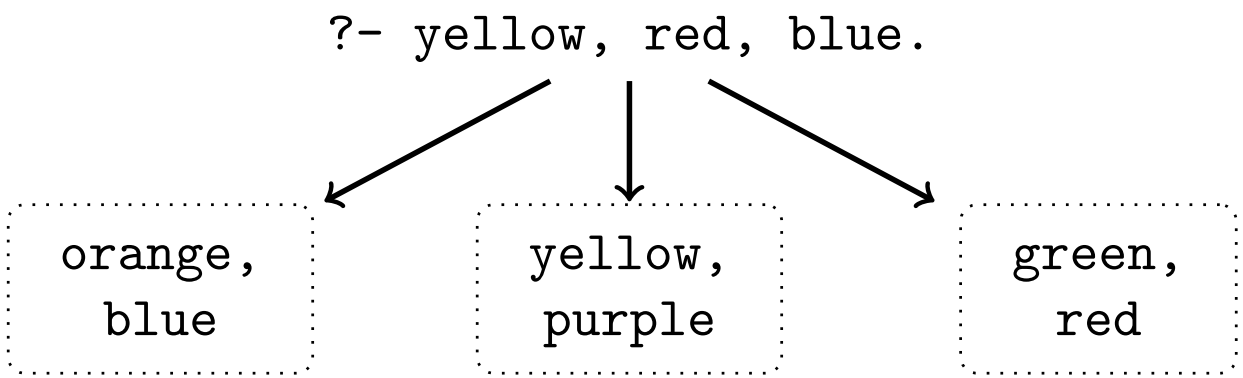
Abstract semantics versus refined Semantics

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



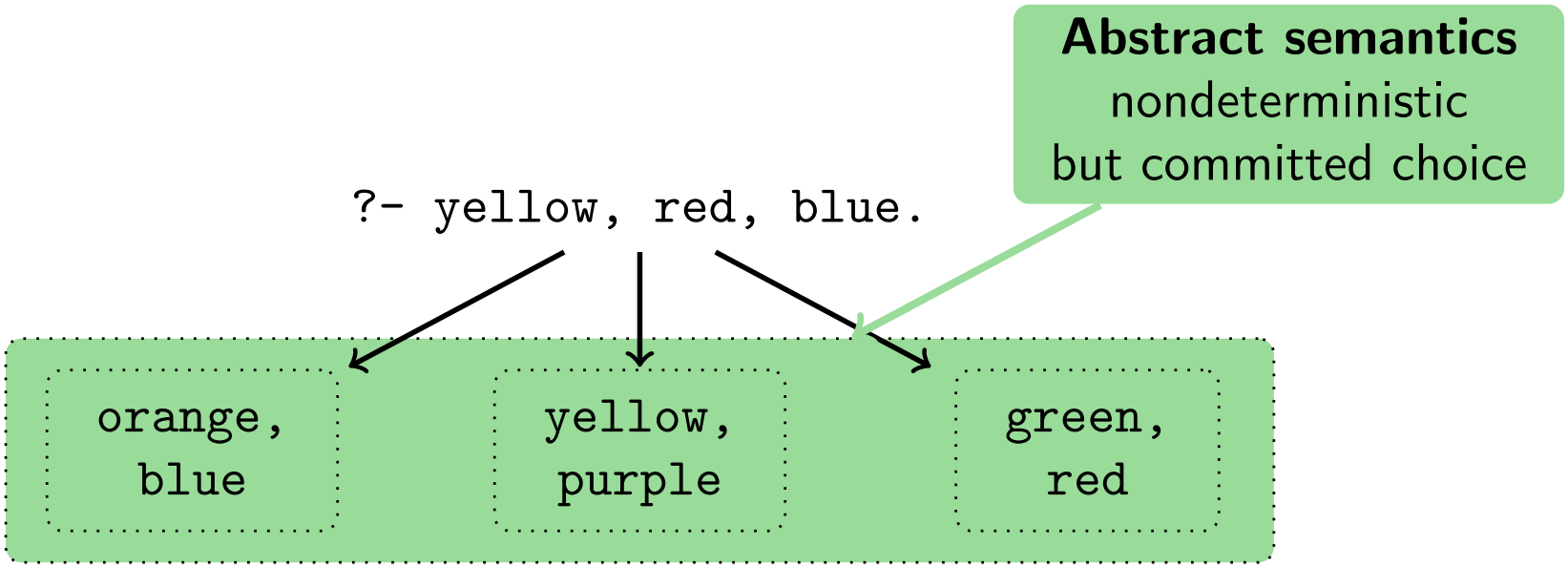
Abstract semantics versus refined Semantics

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



Abstract semantics versus refined Semantics

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

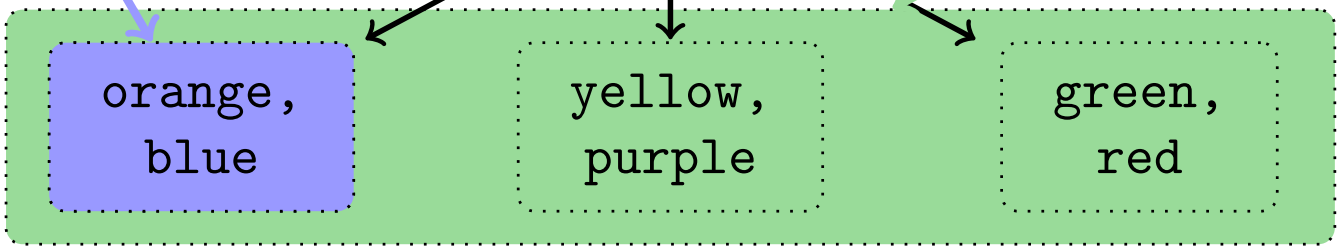


Abstract semantics versus refined Semantics

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

Refined semantics
query: left to right
rule: top to bottom

Abstract semantics
nondeterministic
but committed choice



Summary

Simplification rule

- `head <=> body`
- `body` replaces `head`.

Two execution levels

- **Asbtract semantics** (High-level)
  non-deterministics but commited-choice.
- **Refined semantics** (Low-level)
  (almost) deterministics and commited-choice.
  process queries from left to right and rules from top to bottom.

Outline

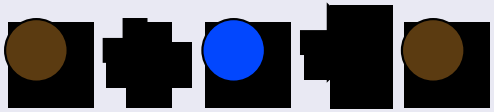
Mixing brown as simplification rules

Brown stays brown.

```
:- chr_constraint brown.                    %
```


```
%
```


```
brown , yellow <=> brown
%
```

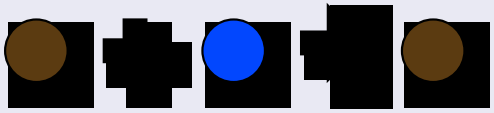

```
brown , red    <=> brown
%
```


```
brown , blue   <=> brown
...
```

Mixing brown as simpagation rules

Brown stays brown.

```
:- chr_constraint brown.                    % 

% 
brown \ yellow <=> true
% 
brown \ red    <=> true
% 
brown \ blue   <=> true
...
```

Mixing brown as simpagation rules

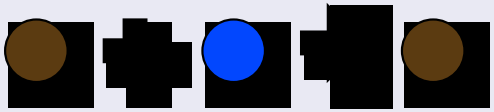
Brown stays brown.

```
:- chr_constraint brown.
```

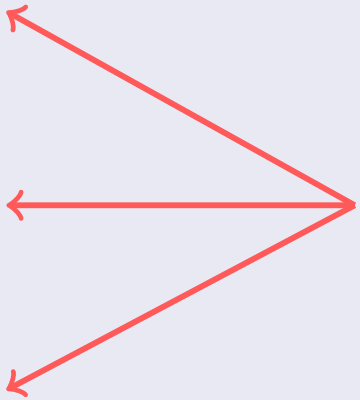
% 

% 
```
brown \ yellow <=> true
```

% 
```
brown \ red    <=> true
```

% 
```
brown \ blue   <=> true
```

```
...
```



Prolog no-op

Mixing brown as simpagation rules

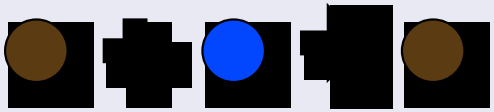
Brown stays brown.

```
:- chr_constraint brown.
```

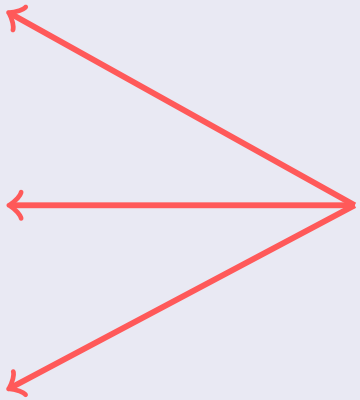
% 

% 
```
brown \ yellow <=> true
```

% 
```
brown \ red    <=> true
```

% 
```
brown \ blue   <=> true
```

```
...
```



Prolog no-op

Simpagation rules : $\texttt{head}_k$ \ $\texttt{head}_r$ <=> body.
"$\texttt{head}_r$ can be replaced by body if $\texttt{head}_k$ is present"

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

?- philosophers_stone, lead
philosophers_stone,
gold

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

?- philosophers_stone, lead
philosophers_stone,
gold

?-philosophers_stone, lead, lead
philosophers_stone,
gold,
gold

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

?- philosophers_stone, lead
philosophers_stone,
gold

?-philosophers_stone, lead, lead
philosophers_stone,
gold,
gold

?-lead, lead
lead,
lead

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

?- philosophers_stone, lead
philosophers_stone,
gold

?-philosophers_stone, lead, lead
philosophers_stone,
gold,
gold

?-lead, lead
lead,
lead

?-lead, lead, philosophers_stone
philosophers_stone,
gold,
gold

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

?- philosophers_stone, lead
philosophers_stone,
gold

?-philosophers_stone, lead, lead
philosophers_stone,
gold,
gold

?-lead, lead
lead,
lead

?-lead, lead, philosophers_stone
philosophers_stone,
gold,
gold

☞ Rules fire until exhaustion of all posibilities !

Transmute lead to gold with simpagation rules

Alchemy

```
philosophers_stone \ lead <=> gold
```

?- philosophers_stone, lead
philosophers_stone,
gold

?-philosophers_stone, lead, lead
philosophers_stone,
gold,
gold

?-lead, lead
lead,
lead

?-lead, lead, philosophers_stone
philosophers_stone,
gold,
gold

- ☞ Rules fire until exhaustion of all posibilities !
- ☞ Constraint stores have a multiset behaviour
  i.e. multiplicity of constraint matters.

Summary

Simplification rule

- `head <=> body`
- replaces `body` with `head`.

Summary

Simplification rule

- `head <=> body`
- replaces `body` with `head`.

Simpagation rule

- $\texttt{head}_k$ `\` $\texttt{head}_r$ `<=> body.`
- replaces $\texttt{head}_r$ with `body`
- if $\texttt{head}_k$ is present.

Summary

Simplification rule

- `head <=> body`
- replaces `body` with `head`.

Simpagation rule

- $\texttt{head}_\texttt{k}$ `\` $\texttt{head}_\texttt{r}$ `<=> body.`
- replaces $\texttt{head}_\texttt{r}$ with `body`
- if $\texttt{head}_\texttt{k}$ is present.

Operationally

- Rules are executed exhaustively.
- Constraint stores have a multiset behaviour.

Outline

Borwn mixing is a little bit tedious

```
brown \ red    <=> true.
brown \ blue   <=> true.
brown \ yellow <=> true.
brown \ purple <=> true.
brown \ green  <=> true.
brown \ brown  <=> true.
```

Borwn mixing is a little bit tedious

```
brown \ red    <=> true.
brown \ blue   <=> true.
brown \ yellow <=> true.
brown \ purple <=> true.
brown \ green  <=> true.
brown \ brown  <=> true.
```

A bit tedious, isn't it ?

Borwn mixing is a little bit tedious

```
brown \ red    <=> true.
brown \ blue   <=> true.
brown \ yellow <=> true.
brown \ purple <=> true.
brown \ green  <=> true.
brown \ brown  <=> true.
```

A bit tedious, isn't it ?

Why not something like this instead ?

```
brown \ _ <=> true.
```

Borwn mixing is a little bit tedious

```
brown \ red    <=> true.
brown \ blue   <=> true.
brown \ yellow <=> true.
brown \ purple <=> true.
brown \ green  <=> true.
brown \ brown  <=> true.
```

A bit tedious, isn't it ?

Why not something like this instead ?

~~`brown \ _ <=> true.`~~ Forbidden!

☞ Constraints within rule must be non-variable !

Mixing color with arguments

```
:- chr_constraint color/1.

color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).
```

Mixing color with arguments

```
:- chr_constraint color/1.

color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).
```

- Constraints cannot be variables

Mixing color with arguments

```
:- chr_constraint color/1.

color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).

color(brown),  color(_)      <=> color(brown).
```

- Constraints cannot be variables
- Constraints can have arguments

Mixing color with arguments

```
:- chr_constraint color/1.

color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).

color(brown),  color(_)      <=> color(brown).
```

- Constraints cannot be variables
- Constraints can have arguments
- Arguments can contain variables

Mixing color with arguments

```
:- chr_constraint color/1.

color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).

color(brown),  color(_)      <=> color(brown).
```

- Constraints cannot be variables
- Constraints can have arguments
- Arguments can contain variables
- Variables in heads are (implicitly) universally quantified

Piggy Bank Merge

```
:- chr_constraint piggy/1.

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

Piggy Bank Merge

```
:- chr_constraint piggy/1.

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

```
?- piggy(5).
piggy(5)
```

Piggy Bank Merge

```
:- chr_constraint piggy/1.

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

```
?- piggy(5).
piggy(5)

?- piggy(5), piggy(1).
piggy(6)
```

Piggy Bank Merge

```
:- chr_constraint piggy/1.

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

```
?- piggy(5).
piggy(5)

?- piggy(5), piggy(1).
piggy(6)

?- piggy(5), piggy(1), piggy(4).
piggy(10)
```

Piggy Bank Merge

```
:- chr_constraint piggy/1.

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

```
?- piggy(5).
piggy(5)

?- piggy(5), piggy(1).
piggy(6)

?- piggy(5), piggy(1), piggy(4).
piggy(10)

?- piggy(5), piggy(1), piggy(4), piggy(2).
piggy(12)
```

Piggy Bank Merge

```
:- chr_constraint piggy/1.

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

```
?- piggy(5).
piggy(5)

?- piggy(5), piggy(1).
piggy(6)

?- piggy(5), piggy(1), piggy(4).
piggy(10)

?- piggy(5), piggy(1), piggy(4), piggy(2).
piggy(12)
```

CHR an embedded language.

- CHR is embedded in a host language.

  Here the host language is Prolog.

- 2-way communication:
  - Prolog calls CHR constraints

    e.g. call from the Prolog top-level
  - CHR (bodies) call Prolog

    e.g. `K is J + 1`

Summary

CHR constraints

- zero or more arguments
- arbitrary Prolog term

Summary

CHR constraints

- zero or more arguments
- arbitrary Prolog term

Rule Heads

- only CHR constraints
- always instantiated

Summary

CHR constraints

- zero or more arguments
- arbitrary Prolog term

Rule Heads

- only CHR constraints
- always instantiated

Rule Bodies

- CHR constraint and
- Host language calls

Outline

Minimum

Problem: remove all non-minimal element from a multiset

?- min(8), min(3), min(6), min(7).
min(3)

Minimum I

```
min(X), min(Y) <=> minimum(X,Y,Z), min(Z).

minimum(X, Y, Z) :- X =< Y, !, X=Z.
minimum(_, Y, Y).
```

Minimum I

```
min(X), min(Y) <=> minimum(X,Y,Z), min(Z).

minimum(X, Y, Z) :- X =< Y, !, X=Z.
minimum(_, Y, Y).
```

```
?- min(8), min(3), min(6), min(7).
min(3)
```

Minimum I

```
min(X), min(Y) <=> minimum(X,Y,Z), min(Z).

minimum(X, Y, Z) :- X =< Y, !, X=Z.
minimum(_, Y, Y).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
```

Minimum I

```
min(X), min(Y) <=> minimum(X,Y,Z), min(Z).

minimum(X, Y, Z) :- X =< Y, !, X=Z.
minimum(_, Y, Y).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
{ERROR: arithmetic:=</2 - instantiation_error}
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

```
?- min(8), min(3), min(6), min(7).
min(3)
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
min(4)
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
min(4)

?- min(X), min(Y)
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
min(4)

?- min(X), min(Y)
min(X), min(Y)
```

Minimum II

```
min(X) , min(Y) <=> X=<Y | min(X).
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
min(4)

?- min(X), min(Y)
min(X), min(Y)
```

☞ If no rule applies, then the execution is delayed

Minimum II

```
min(X) \ min(Y) <=> X=<Y | true.
```

```
?- min(8), min(3), min(6), min(7).
min(3)

?- min(X), min(Y), X=4, Y=5.
min(4)

?- min(X), min(Y)
min(X), min(Y)
```

☞ If no rule applies, then the execution is delayed

Summary

Simplification rule

- `head <=> guard | body`
- replaces `body` with `head`
- if guards is true

Summary

Simplification rule

- `head <=> guard | body`
- replaces `body` with `head`
- if guards is true

Simpagation rule

- $\texttt{head}_\texttt{k}$ `\` $\texttt{head}_\texttt{r}$ `<=> guard | body.`
- replaces $\texttt{head}_\texttt{r}$ with `body`
- if $\texttt{head}_\texttt{k}$ is present and
- guards is true

Summary

Simplification rule

- `head <=> guard | body`
- replaces `body` with `head`
- if guards is true

Simpagation rule

- $\texttt{head}_\texttt{k}$ `\` $\texttt{head}_\texttt{r}$ `<=> guard | body.`
- replaces $\texttt{head}_\texttt{r}$ with `body`
- if $\texttt{head}_\texttt{k}$ is present and
- guards is true

Operationally

- if no rule applies, then the execution is delayed.

Outline

Color matching

```
color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).
color(brown) \ color(_)      <=> true.
```

Color matching

```
color(yellow), color(red)    <=> color(orange).
color(red),    color(blue)   <=> color(purple).
color(blue),   color(yellow) <=> color(green).
color(brown) \ color(_)      <=> true.
```

Matching:

- "inline" matching notation for guard
- equivalent meaning as explicit guard

Meaning of matching

Matching	Guard

Meaning of matching

Matching	**Guard**
`c(X) <=> true.`	`c(X) <=> true	true.`

Meaning of matching

Matching	**Guard**
`c(X) <=> true.`	`c(X) <=> true | true.`
`c(a) <=> true.`	`c(X) <=> X==a | true.`

Meaning of matching

Matching	**Guard**
`c(X) <=> true.`	`c(X) <=> true	true.`
`c(a) <=> true.`	`c(X) <=> X==a	true.`
`c(f(A)) <=> true.`	`c(X) <=> nonvar(X), X=f(A)	true.`

Meaning of matching

Matching	**Guard**
`c(X) <=> true.`	`c(X) <=> true	true.`
`c(a) <=> true.`	`c(X) <=> X==a	true.`
`c(f(A)) <=> true.`	`c(X) <=> nonvar(X), X=f(A)	true.`
`c(X,X) <=> true.`	`c(X,Y) <=> X == Y	true.`

Meaning of matching

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

Summary

Matching

- short-hand for equality-based guards
- one-way unification
  - only instantiates rule heads
  - does not instantiate constraint store.

Outline

A more complexe program I

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

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

?- generate(4)

A more complexe program I

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

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

?- generate(4)
value(10)

?- generate(5)

A more complexe program I

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

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

?- generate(4)
value(10)

?- generate(5)
value(15)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)
value(7), generate(2)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)
value(7), generate(2)
value(7), value(2), generate(1)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)
value(7), generate(2)
value(7), value(2), generate(1)
value(9), generate(1)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)
value(7), generate(2)
value(7), value(2), generate(1)
value(9), generate(1)
value(9), value(1), generate(0)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)
value(7), generate(2)
value(7), value(2), generate(1)
value(9), generate(1)
value(9), value(1), generate(0)
value(10), generate(0)

A more complexe example I

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

value(I), value(J) <=> K is J + 1, value(K).
```

generate(4)
value(4), generate(3)
value(4), value(3), generate(2)
value(7), generate(2)
value(7), value(2), generate(1)
value(9), generate(1)
value(9), value(1), generate(0)
value(10), generate(0)
value(10)

A more complexe program II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(X)

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

?- generate(4)

A more complexe program II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(X)

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

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

?- generate(10)

A more complexe program II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(X)

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

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

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

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(10),
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(9),
```

```
                        value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(8),
```

```
          value(9), value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(7),
```

```
value(8), value(9), value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(6),

                    value(7),
value(8), value(9), value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(5),

          value(6), value(7),
value(8), value(9), value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(4),

value(5), value(6), value(7),
value(8), value(9), value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(4),

value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(3),
                    value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(3),
                    value(4),
value(5), value(6), value(7),
~~value(8)~~, value(9), ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(2),
          value(3), value(4),
value(5), value(6), value(7),
~~value(8)~~, value(9), ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(2),
          value(3), value(4),
value(5), value(6), value(7),
~~value(8)~~, ~~value(9)~~, ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(2),
          value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(1),
value(2), value(3), value(4),
value(5), ~~value(6)~~, value(7),
~~value(8)~~, ~~value(9)~~, ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

```
generate(1),
value(2), value(3), ~~value(4)~~,
value(5), ~~value(6)~~, value(7),
~~value(8)~~, ~~value(9)~~, ~~value(10)~~
```

A more complexe example II

```
generate(1) <=> true.
generate(X) <=> X > 1 | value(X),
                        Y is X - 1, generate(Y).

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

value(2), value(3), ~~value(4)~~,
value(5), ~~value(6)~~, value(7),
~~value(8)~~, ~~value(9)~~, ~~value(10)~~

Summary

Programs

- CHR programs consist of sequence of rules
- Query are traversed form left to right
- Rules are traversed top to bottom
- Rest of constraint waits

Outline

Special cases of simpagation rules

Simpagation rule

$\texttt{head}_k$ \ $\texttt{head}_r$ <=> guard | body.

Special cases of simpagation rules

Simpagation rule

$\texttt{head}_\texttt{k}$ \ $\texttt{head}_\texttt{r}$ <=> guard | body.

If $\texttt{head}_\texttt{r} = \emptyset$:

Special cases of simpagation rules

Simpagation rule

$\texttt{head}_\texttt{k}$ \ $\texttt{head}_\texttt{r}$ <=> guard | body.

If $\texttt{head}_\texttt{r} = \emptyset$:

Simplification rule

$\texttt{head}_\texttt{k}$ <=> guard | body.

Special cases of simpagation rules

Simpagation rule

$\texttt{head}_\texttt{k}$ \ $\texttt{head}_\texttt{r}$ <=> guard | body.

If $\texttt{head}_\texttt{r} = \emptyset$:

Simplification rule

$\texttt{head}_\texttt{k}$ <=> guard | body.

If $\texttt{head}_\texttt{r} = \emptyset$:

Special cases of simpagation rules

Simpagation rule

$\texttt{head}_k$ \ $\texttt{head}_r$ <=> guard | body.

If $\texttt{head}_r = \emptyset$:

Simplification rule

$\texttt{head}_k$ <=> guard | body.

If $\texttt{head}_r = \emptyset$:

Propagation rule

$\texttt{head}_k$ ==> guard | body.

- adds body
- if $\texttt{head}_k$ is present and
- guard is true

Propagation 1

```
a, b ==> c
```

?- a, b.

Propagation 1

```
a, b ==> c
```

?- a, b.

a,

b,

c

Propagation 1

```
a, b ==> c
```

?- a, b.

a,

b,

c

?- a, b, b.

Propagation 1

```
a, b ==> c
```

?- a, b.

a,

b,

c

?- a, b, b.

a,

b,

b,

c,

c

Propagation 1

```
a, b ==> c
```

?- a, b.
a,
b,
c

?- a, b, b.
a,
b,
b,
c,
c

☞ Propagation rules apply only once on each occurence of their head!

Propagation 1

```
a, b ==> c
```

?- a, b.
a,
b,
c

?- a, b, b.
a,
b,
b,
c,
c

?- a, a, b, b.

☞ Propagation rules apply only once on each occurence of their head!

Propagation 1

```
a, b ==> c
```

?- a, b.
a,
b,
c

?- a, b, b.
a,
b,
b,
c,
c

?- a, a, b, b.
a,
a,
b,
b,
c,
c,
c,
c

☞ Propagation rules apply only once on each occurence of their head!

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

generate(10)

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
value(2)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

generate(10),
value(2)

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
value(2), value(3)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
value(2), value(3)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
value(2), value(3), value(4)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
~~value(2)~~, value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~,
value(2)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

generate(10),
~~value(2)~~, value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~,
value(2)

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
~~value(2)~~, value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~,
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
~~value(2)~~, value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~,
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

Propagation 2

```
generate(N) ==> value(2).
generate(N), value(I) ==> I < N | J is I + 1, value(J).

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

```
?- generate(10).
ERROR: Out of local stack
```

```
generate(10),
~~value(2)~~, value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~,
~~value(2)~~, value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~
value(2), ...
```

Propagation 3

```
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.
```

Propagation 3

```
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).
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)
```

generate(10)

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2), value(3)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2), value(3)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2), value(3), value(4)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)
```

```
generate(10),
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), value(10)
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)

generate(10),
value(2), value(3), value(4),
value(5), value(6), value(7),
value(8), value(9), ~~value(10)~~
```

Propagation 3

```
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).
generate(10),
value(2),
value(3),
value(5),
value(7)
```

```
generate(10),
value(2), value(3), ~~value(4)~~,
value(5), ~~value(6)~~, value(7),
~~value(8)~~, ~~value(9)~~, ~~value(10)~~
```

Summary

Simplification rule : `head <=> guard | body`

- replaces `head` with `body`
- if guards is true

Simpagation rule : $\texttt{head}_k$ \ $\texttt{head}_r$ `<=> guard | body.`

- replaces $\texttt{head}_r$ with `body`
- if $\texttt{head}_k$ is present and
- guards is true

Propagation rule : `head ==> guard | body.`

- adds `body` (only once)
- if `head` is present and
- guards is true

Summary

Simplification rule : `head <=> guard | body`

- replaces `head` with `body`
- if guards is true

Simpagation rule : $\texttt{head}_k$ \ $\texttt{head}_r$ <=> guard | body.

- replaces $\texttt{head}_r$ with `body`
- if $\texttt{head}_k$ is present and
- guards is true

Propagation rule : `head ==> guard | body.`

- adds `body` (only once)
- if `head` is present and
- guards is true

☞ simpagation are usefull to avoid trivial non-termination problems

Outline

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

```
?- c(hello), c(world).
```

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

```
?- c(hello), c(world).
hello
world
```

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

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

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

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

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

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

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

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

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

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

☞ constraints supend in the store.

Hello World!

```
c(hello) <=> write(hello), nl.
c(world) => write(world), nl.
```

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

- ☞ constraints supend in the store.
- ☞ unification reactivates supendend constraints.

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
```

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
21
```

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
21
```

In CHR:

```
chr_freeze(X, G) <=> nonvar(X) | call(G).
```

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
21
```

In CHR:

```
chr_freeze(X, G) <=> nonvar(X) | call(G).
```

```
?- chr_freeze(X, write(X)), write(2), nl.
```

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
21
```

In CHR:

```
chr_freeze(X, G) <=> nonvar(X) | call(G).
```

```
?- chr_freeze(X, write(X)), write(2), nl.
2
chr_freeze(X, write(X))
```

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
21
```

In CHR:

```
chr_freeze(X, G) <=> nonvar(X) | call(G).
```

```
?- chr_freeze(X, write(X)), write(2), nl.
2
chr_freeze(X, write(X))
?- chr_freeze(X, write(X)), write(2), X=1, nl.
```

Coroutinings in CHR

In usual prolog systems:

```
?- freeze(X, write(X)), write(2), X=1, nl.
21
```

In CHR:

```
chr_freeze(X, G) <=> nonvar(X) | call(G).
```

```
?- chr_freeze(X, write(X)), write(2), nl.
2
chr_freeze(X, write(X))
?- chr_freeze(X, write(X)), write(2), X=1, nl.
21
```

Inequality Constraints in CHR

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

?- neq(a,a).

Inequality Constraints in CHR

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

?- neq(a,a).

no

Inequality Constraints in CHR

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

?- neq(a,a).
no

?- neq(a,b).

Inequality Constraints in CHR

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

?- neq(a,a).
no

?- neq(a,b).
yes

Inequality Constraints in CHR

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

?- neq(a,a).
no

?- neq(a,b).
yes

?- neq(A,B).

Inequality Constraints in CHR

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

?- neq(a,a).
no

?- neq(a,b).
yes

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

Inequality Constraints in CHR

```
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.

Inequality Constraints in CHR

```
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

Inequality Constraints in CHR

```
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.

Inequality Constraints in CHR

```
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.
neq(a,B)

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

Inequality Constraints in CHR

```
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.
neq(a,B)

?- neq(A,B), A=a, B=a.
no

Inequality Constraints in CHR

```
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.
neq(a,B)

?- neq(A,B), A=a, B=a.
no

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

Inequality Constraints in CHR

```
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.
neq(a,B)

?- neq(A,B), A=a, B=a.
no

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

Inequality Constraints in CHR

```
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.
neq(a,B)

?- 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).

Inequality Constraints in CHR

```
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.
neq(a,B)

?- 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))

Inequality Constraints in CHR

```
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.
neq(a,B)

?- 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))

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

Inequality Constraints in CHR

```
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.
neq(a,B)

?- 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))

?- neq(A,B), A=f(C), B=f(D), C=D.
no

Inequality Constraints in CHR

```
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.
neq(a,B)

?- 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))

?- neq(A,B), A=f(C), B=f(D), C=D.
no

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

Inequality Constraints in CHR

```
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.
neq(a,B)

?- 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))

?- neq(A,B), A=f(C), B=f(D), C=D.
no

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

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
yes
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
yes

?- domain(X, 1, 5), domain(X, 3, 8)
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
yes

?- domain(X, 1, 5), domain(X, 3, 8)
domain(X, 3, 5)
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
yes

?- domain(X, 1, 5), domain(X, 3, 8)
domain(X, 3, 5)

?- domain(X, 1, 5), domain(X, 8, 15)
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
yes

?- domain(X, 1, 5), domain(X, 3, 8)
domain(X, 3, 5)

?- domain(X, 1, 5), domain(X, 8, 15)
no
```

Finite Domain Solver in CHR

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).
```

```
?- domain(X, 1, 5)
domain(X, 1, 5)

?- domain(X, 1, 5), X = 3
yes

?- domain(X, 1, 5), domain(X, 3, 8)
domain(X, 3, 5)

?- domain(X, 1, 5), domain(X, 8, 15)
no
```

Summary

Reactivation

- rules may not fire because of lack of instantiation
- upon instantiation, the rule may now fire
- useful for implementing constraint solvers

Outline

Summary: CHR vs Prolog

	Prolog	CHR
head	1	≤ 1
rule selection	unification	matching & guard
different rules	backtracking	committed choice
no rule	failure	delay

Backtracking for Labeling

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).

label(X) <=> ground(X) | true.
label(X)  domain(X, Min, Max) <=>
        Max_ is (Min + Max) // 2, domain(X, Min, Max_) ;
        Min_ is (Min + Max) // 2 + 1, domain(X, Min_, Max).
```

Backtracking for Labeling

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).

label(X) <=> ground(X) | true.
label(X)  domain(X, Min, Max) <=>
        Max_ is (Min + Max) // 2, domain(X, Min, Max_) ;
        Min_ is (Min + Max) // 2 + 1, domain(X, Min_, Max).
```

?- domain(X, 1, 3), label(X).

Backtracking for Labeling

```
domain(X, Min, Max) <=> ground(X) | Min =< X, X =< Max.
domain(X, Min, Max) <=> Min >= Max | X = Min, X = Max.
domain(X, Min1, Max1), domain(X, Min2, Max2) <=>
        max(Min1, Min2, Min), min(Max1, Max2, Max),
        domain(X, Min, Max).

label(X) <=> ground(X) | true.
label(X)  domain(X, Min, Max) <=>
        Max_ is (Min + Max) // 2, domain(X, Min, Max_) ;
        Min_ is (Min + Max) // 2 + 1, domain(X, Min_, Max).
```

```
?- domain(X, 1, 3), label(X).
X = 1 ;
X = 2 ;
X = 3 ;
no
```

Questions ?