Computational Logic

Constraint Logic Programming
Constraints

• Born within AI: e.g. house design

• Constraints used as problem representation:

  *The man in yellow does not have green eyes*
  *The murderer knows no detective will ever wear dark clothes*

• A solution is an assignment which agrees with the initial constraints:

  *Murderer: López, green eyes, Magnum gun*

• Or, alternatively, the solution can also be a set of constraints:

  *The murderer is one of those who had met the cabaret entertainer*
  *(they represent several ground mappings from elements to variables)*

• There may be no solution:

  *Natural death*
A General View

- Ancestors:
  - SKETCHPAD (1963), Waltz’s algorithm (1965?), THINGLAB (1981), MACSYMA (1983), ...

- Constraints in logic languages – the origin of “constraint programming”:
  - General theory developed (Jaffar and Lassez ’97).
  - First, standalone systems developed: clpr, CHIP, ...
  - Now included in many Prologs (e.g., clpr/clpq/clpfd packages in Ciao).

- Constraints in imperative languages:
  - Equation solving libraries (ILOG, GECODE, ...)
  - Timestamping of variables: $x := x + 1 \leftrightarrow x_{i+1} := x_i + 1$
    (similar to iterative methods in numerical analysis)

- Constraints in functional languages, via extensions:
  - Evaluation of expressions including free variables.
  - *Absolute Set Abstraction.*
A comparison with classic LP (I)

- Example (plain Prolog):
  \[ q(X, Y, Z) :\neg Z = f(X, Y). \]

\[
\text{?- q(3, 4, Z).} \\
Z = f(3,4)
\]

\[
\text{?- q(X, Y, f(3,4)).} \\
X = 3, Y = 4
\]

\[
\text{?- q(X, Y, Z).} \\
Z = f(X, Y)
\]

- Example (plain Prolog):
  \[ p(X, Y, Z) :\neg Z \text{ is } X + Y. \]

\[
\text{?- p(3, 4, Z).} \\
Z = 7
\]

\[
\text{?- p(X, 4, 7).} \\
\text{\{INSTANTIATION ERROR\} \leftarrow is/2 not reversible, does not work!}
\]
A Comparison with classic LP (II)

- Example (**CLP(R) package**):

```prolog
:- use_package(clpr).
p(X, Y, Z) :- Z =. X + Y.
?- p(3, 4, Z).
Z =. 7

?- p(X, 4, 7).
X =. 3

4 ?- p(X, Y, 7).
X =. 7 - Y ← with clpr arithmetic is reversible!
```
A Comparison with classic LP (III)

- Features in CLP:
  - Domain of computation (reals, integers, booleans, etc).
    Have to meet some conditions.
  - Type of constraints allowed for each domain: e.g. arithmetic constraints
    ($+, *, =, \leq, \geq, <, >$)
  - Constraint solving algorithms: simplex, gauss, etc.

- Classical LP can be viewed as a constraint logic language over Herbrand terms
  with a single constraint predicate symbol: $=$
A Comparison with classic LP (IV)

• Advantages:
  ◦ Helps making programs expressive and flexible.
  ◦ May save much coding.
  ◦ In some cases, more efficient than classic LP programs due to solvers typically being very efficiently implemented.
  ◦ Also, efficiency due to search space reduction:
    * LP: generate-and-test.
    * CLP: constrain-and-generate.

• Disadvantages:
  ◦ Complexity of solver algorithms (simplex, gauss, etc) can affect performance.

• Solutions:
  ◦ better algorithms
  ◦ compile-time optimizations (program transformation, global analysis, etc)
  ◦ parallelism
Example of Search Space Reduction

- Using **plain Prolog** (generate–and–test):
  ```prolog
  % Find three consecutive numbers in the p/1 relation.
  solution(X, Y, Z) :-
      p(X), p(Y), p(Z),
      test(X, Y, Z).
  test(X, Y, Z) :- Y is X + 1, Z is Y + 1.
  ```

- Query:
  ```prolog
  ?- solution(X, Y, Z).
  X = 14, Y = 15, Z = 16 ? ;
  no
  ```

- 458 steps (all solutions: 465 steps).
Example of Search Space Reduction

- Using the **CLP(ℜ) package** (generate–and–test):

```
% Find three consecutive numbers in the p/1 relation.
:- use_package(clpr).
solution(X, Y, Z) :-
    p(X), p(Y), p(Z),
    test(X, Y, Z).


test(X, Y, Z) :- Y == X + 1, Z == Y + 1.
```

- Query:

```
?- solution(X, Y, Z).
X == 14, Y == 15, Z == 16 ? ;
no
```

- 458 steps (all solutions: 465 steps).
Generate–and–test Search Tree

A

B

X=14
X=15
X=16
X=7
X=3
X=11

A5
A4
A3
A2
A1

Y=14
Y=15
Y=16
Y=7
Y=3
Y=11

B5
B4
B3
B2
B1

Z=14
Z=15
Z=16
Z=7
Z=3
Z=11
Example of Search Space Reduction

- **Move** \( \text{test}(X, Y, Z) \) **to the beginning (constrain–and–generate):**

  \[
  \text{% Find three consecutive numbers in the p/1 relation.} \\
  :- \text{use\_package}\text{(clpr)}. \\
  \text{solution}(X, Y, Z) :- \\
  \text{test}(X, Y, Z), \\
  \text{p}(X), \text{p}(Y), \text{p}(Z). \\
  \]

- **Using plain Prolog:** \( \text{test}(X, Y, Z) :- Y \text{ is } X + 1, Z \text{ is } Y + 1. \)

  \[
  \text{?- solution}(X, Y, Z). \\
  \{\text{INSTANTIATION ERROR}\}
  \]

- **Using the CLP(\(\mathbb{R}\)) package:** \( \text{test}(X, Y, Z) :- Y \text{ =}_.X + 1, Z \text{ =}_.Y + 1. \)

  \[
  \text{?- solution}(X, Y, Z). \\
  X \text{ =}_.14, Y \text{ =}_.15, Z \text{ =}_.16 ? ; \\
  \text{no}
  \]

In **6 steps** (and all solutions in **11 steps**)!
Constrain–and–generate Search Tree

\[ Y = 15 \quad X = 15 \quad X = 16 \quad X = 7 \quad X = 3 \quad X = 11 \]

\[ Z = 16 \]

\[ g \]

\[ Y = 16 \]

\[ Z = 16 \]
Constraint Domains

- Semantics parameterized by the constraint domain: CLP(\mathcal{X})$, where $\mathcal{X} \equiv (\Sigma, D, \mathcal{L}, T)$
- Signature $\Sigma$: set of predicate and function symbols, together with their arity
- $\mathcal{L} \subseteq \Sigma$–formulae: constraints
- $D$ is the set of actual elements in the domain
- $\Sigma$–structure $\mathcal{D}$: gives the meaning of predicate and function symbols (and hence, constraints).
- $T$ a first–order theory (axiomatizes some properties of $\mathcal{D}$)
- $(\mathcal{D}, \mathcal{L})$ is a constraint domain
- Assumptions:
  - $\mathcal{L}$ built upon a first–order language
  - $\in \Sigma$ is identity in $\mathcal{D}$
  - There are identically false and identically true constraints in $\mathcal{L}$
  - $\mathcal{L}$ is closed w.r.t. renaming, conjunction and existential quantification
Domains (I)

- $\Sigma = \{0, 1, +, *, =, <, \leq\}$, $D = \mathbb{R}$, $\mathcal{D}$ interprets $\Sigma$ as usual, $\mathcal{R} = (\mathcal{D}, \mathcal{L})$

  $\rightarrow$ **Arithmetic over the reals** ("$\mathcal{R}$" domain).
  - Eg.: $x^2 + 2xy < \frac{y}{x} \land x > 0$ ($\equiv xxx + xxy + xxy < y \land 0 < x$)
  - Question: is 0 needed? How can it be represented?

- $\Sigma' = \{0, 1, +, =, <, \leq\}$, $\mathcal{R}_{Lin} = (\mathcal{D}', \mathcal{L}')$

  $\rightarrow$ **Linear arithmetic** ("$\mathcal{R}_{Lin}$" domain)
  - Eg.: $3x - y < 3$ ($\equiv x + x + x < 1 + 1 + 1 + y$)

- $\Sigma'' = \{0, 1, +, =\}$, $\mathcal{R}_{LinEq} = (\mathcal{D}'', \mathcal{L}'')$

  $\rightarrow$ **Linear equations** ("$\mathcal{R}_{LinEq}$" domain)
  - Eg.: $3x + y = 5 \land y = 2x$
Domains (II)

- A very special domain:
  - $\Sigma = \{ <\text{constant and function symbols}>, = \}$
  - $D = \{ \text{finite trees} \}$
  - $D$ interprets $\Sigma$ as tree constructors
  - Each $f \in \Sigma$ with arity $n$ maps $n$ trees to a tree with root labeled $f$ and whose subtrees are the arguments of the mapping
  - Constraints: syntactic tree equality
  - $\mathcal{FT} = (D, \mathcal{L})$

  → **Equality constraints over the Herbrand domain** ($\mathcal{FT}$ domain)
    - Eg.: $g(h(Z), Y) = g(Y, h(a))$

- $\text{LP} \equiv \text{CLP}(\mathcal{FT})$
Domains (III)

- $\Sigma = \{<\text{constants}>, \lambda, ., ::, =\}$
- $D = \{\text{finite strings of constants}\}$
- $D$ interprets $\cdot$ as string concatenation, $::$ as string length

→ Equations over strings of constants ($D$ domain)
  ◇ Eg.: $X.A.X = X.A$

- $\Sigma = \{0, 1, \neg, \land, =\}$
- $D = \{true, false\}$
- $D$ interprets symbols in $\Sigma$ as boolean functions
- $BOOL = (D, L)$

→ Boolean constraints) ($BOOL$ domain)
  ◇ Eg.: $\neg(x \land y) = 1$
CLP(\mathcal{X}) Programs

- Recall that:
  - $\Sigma$ is a set of predicate and function symbols
  - $\mathcal{L} \subseteq \Sigma$ – formulae are the constraints
- $\Pi$: set of predicate symbols definable by a program
- Atom: $p(t_1, t_2, \ldots, t_n)$, where $t_1, t_2, \ldots, t_n$ are terms and $p \in \Pi$
- Primitive constraint: $p(t_1, t_2, \ldots, t_n)$, where $t_1, t_2, \ldots, t_n$ are terms and $p \in \Sigma$ is a predicate symbol
- Every constraint is a (first–order) formula built from primitive constraints
- The class of constraints will vary (generally only a subset of formulas are considered constraints)
- A CLP program is a collection of rules of the form $a \leftarrow b_1, \ldots, b_n$ where $a$ is an atom and the $b_i$’s are atoms or constraints
- A fact is a rule $a \leftarrow c$ where $c$ is a constraint
- A goal (or query) $G$ is a conjunction of constraints and atoms
A case study: CLP($\mathbb{R}$)

- CLP($\mathbb{R}$) is a language based on Prolog, with the addition of constraint solving capabilities over the reals ($\mathbb{R}_{Lin}$)
  - Uses the same execution strategy as standard Prolog (depth-first, left-to-right)
  - Is able to solve directly linear (dis)equations over the reals
  - Non-linear equations are delayed, waiting for them to eventually become linear
  - Most relevant feature w.r.t. Prolog (for our purposes): `is/2` disappears, and is subsumed by `=/2` and (extended) unification
- Note: CLP($\mathbb{R}$) is really CLP(($\mathbb{R}$, $\mathcal{FT}$)) — $\mathcal{FT}$ is often omitted.

- Supported in Ciao via the `clpr` package.
  - Uses `.=`., `.>`, etc. to distinguish the `clpr` constraints from the ISO-Prolog arithmetic primitives.
Linear Equations (CLP(R) package)

• Vector × vector multiplication (dot product):
  \[ \cdot : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R} \]
  \[(x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n) = x_1 \cdot y_1 + \cdots + x_n \cdot y_n\]

• Vectors represented as lists of numbers

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prod([], [], Result) :- Result .==. 0.</td>
<td>?- prod([], [], Result).</td>
</tr>
<tr>
<td>prod([X</td>
<td>Xs], [Y</td>
</tr>
</tbody>
</table>
  Result .==. X * Y + Rest, |
  prod(Xs, Ys, Rest). |

• Unification becomes constraint solving!

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?- prod([2, 3], [4, 5], K).</td>
<td></td>
</tr>
</tbody>
</table>
  K .==. 23 |
| ?- prod([2, 3], [5, X2], 22). |
  X2 .==. 4 |
| ?- prod([2, 7, 3], [Vx, Vy, Vz], 0). |
  Vx .==. -1.5*Vz - 3.5*Vy |

• Any computed answer is, in general, an equation over the variables in the query
Systems of Linear Equations (CLP($\mathbb{R}$))

- Can we solve systems of equations? E.g.,
  
  \[
  3x + y = 5 \\
  x + 8y = 3
  \]

- Write them down at the top level prompt:

```
?- prod([3, 1], [X, Y], 5), prod([1, 8], [X, Y], 3).
X .=. 1.6087, Y .=. 0.173913
```

- A more general predicate can be built mimicking the mathematical vector notation $A \cdot x = b$:

```
system(_Vars, [], []).  
system(Vars, [Co|Coefs], [Ind|Indeps]) :-  
    prod(Vars, Co, Ind),  
    system(Vars, Coefs, Indeps).
```

- We can now express (and solve) equation systems

```
?- system([X, Y], [[3, 1],[1, 8]],[5, 3]).  
X .=. 1.6087, Y .=. 0.173913
```
Non–linear Equations (CLP(ℜ))

- Non–linear equations are delayed

\[- \sin(X) \,.=\, \cos(X)\]
\[\sin(X) \,.=\, \cos(X)\]

- This is also the case if there exists some procedure to solve them

\[- X^x + 2x^x + 1 \,.=\, 0 \]
\[- 2x - 1 \,.=\, x^x x\]

- Reason: no general solving technique is known. CLP(ℜ) solves only linear (dis)equations.

- Once equations become linear, they are handled properly:

\[- X \,.=\, \cos(\sin(Y)), Y \,.=\, 2+y*3, Y \,.=\, -1, X \,.=\, 0.666367\]

- Disequations are solved using a modified, incremental Simplex

\[- X + Y \,.\leq\, 4, Y \,.\geq\, 4, X \,.\geq\, 0 \]
\[Y \,.=\, 4, X \,.=\, 0\]
Fibonacci Revisited (Prolog)

• Fibonacci numbers:

\[
F_0 = 0 \\
F_1 = 1 \\
F_{n+2} = F_{n+1} + F_n
\]

• (The good old) Prolog version:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

• Can only be used with the first argument instantiated to a number
Fibonacci Revisited (CLP(ℜ))

- CLP(ℜ) package version: syntactically similar to the previous one:

```prolog
fib(N,N) :- N .=. 0.
fib(N,N) :- N .=. 1.
fib(N,R) :- N >. 1, F1 >=. 0, F2 >=. 0,
           N1 =. N - 1, N2 =. N - 2,
           fib(N1,F1), fib(N2,F2),
           R =. F1 + F2.
```

- Note all constraints included in program (F1 >= 0, F2 >= 0) – good practice!
- Only real numbers and equations used (no data structures, no other constraint system): “pure CLP(ℜ)”
- Semantics greatly enhanced! E.g.

```
?- fib(N, F).
F =. 0, N =. 0 ;
F =. 1, N =. 1 ;
F =. 1, N =. 2 ;
F =. 2, N =. 3 ;
F =. 3, N =. 4 ;
```
Analog RLC circuits (CLP(ℜ))

- Analysis and synthesis of analog circuits
- RLC network in steady state
- Each circuit is composed either of:
  - A simple component, or
  - A connection of simpler circuits
- For simplicity, we will suppose subnetworks connected only in parallel and series → Ohm’s laws will suffice (other networks need global, i.e., Kirchoff’s laws)
- We want to relate the current (I), voltage (V) and frequency (W) in steady state
- Entry point: \texttt{circuit(C, V, I, W)} states that:
  - across the network \( C \), the voltage is \( V \), the current is \( I \) and the frequency is \( W \)
- \( V \) and \( I \) must be modeled as complex numbers (the imaginary part takes into account the angular frequency)
- Note that Herbrand terms are used to provide data structures
Analog RLC circuits (CLP(ℜ))

- Complex number $X + Yi$ modeled as $c(X, Y)$

- Basic operations:
  
  ```prolog
  c_add(c(Re1, Im1), c(Re2, Im2), c(Re12, Im12)) :-
  Re12 =. Re1 + Re2,
  Im12 =. Im1 + Im2.
  
  c_mult(c(Re1, Im1), c(Re2, Im2), c(Re3, Im3)) :-
  Re3 =. Re1 * Re2 - Im1 * Im2,
  Im3 =. Re1 * Im2 + Re2 * Im1.
  ```

(equality is $c_equal(c(R, I), c(R, I))$, can be left to [extended] unification)
Circuits in series:

\[
\text{circuit}\left(\text{series}(N1, N2), V, I, W\right) \leftarrow \text{c_add}(V1, V2, V), \text{circuit}(N1, V1, I, W), \text{circuit}(N2, V2, I, W).
\]

Circuits in parallel:

\[
\text{circuit}\left(\text{parallel}(N1, N2), V, I, W\right) \leftarrow \text{c_add}(I1, I2, I), \text{circuit}(N1, V, I1, W), \text{circuit}(N2, V, I2, W).
\]
Analog RLC circuits (CLP(ℜ))

Each basic component can be modeled as a separate unit:

- **Resistor:** \( V = I \ast (R + 0i) \)

```prolog
circuit(resistor(R), V, I, _W) :-
c_mult(I, c(R, 0), V).
```

- **Inductor:** \( V = I \ast (0 + WLi) \)

```prolog
circuit(inductor(L), V, I, W) :-
    Im =. W * L,
c_mult(I, c(0, Im), V).
```

- **Capacitor:** \( V = I \ast (0 - \frac{1}{WC}i) \)

```prolog
circuit(capacitor(C), V, I, W) :-
    Im =. -1 / (W * C),
c_mult(I, c(0, Im), V).
```
Analog RLC circuits ($\mathcal{CLP}(\mathbb{R})$)

- Example:

\begin{align*}
\begin{array}{c}
\text{R} = ? \\
\text{C} = ? \\
\text{V} = 4.5 \\
\text{I} = 0.65 \\
\text{L} = 0.073 \\
\text{\omega} = 2400
\end{array}
\end{align*}

?- circuit(parallel(inductor(0.073),
series(capacitor(C), resistor(R))),
c(4.5, 0), c(0.65, 0), 2400).

R = 6.91229, C = 0.00152546

?- circuit(C, c(4.5, 0), c(0.65, 0), 2400).
The N Queens Problem

• Problem:
  place \( \text{N} \) chess queens in a \( \text{N} \times \text{N} \) board such that they do not attack each other

• Data structure: a list holding the column position for each row

• The final solution is a permutation of the list \([1, 2, \ldots, \text{N}]\)

• E.g.: the solution \( \begin{array}{|c|c|c|c|}
1 & 2 & 3 & 4 \\
4 & 1 & 3 & 2 \\
3 & 4 & 2 & 1 \\
2 & 3 & 1 & 4 \\
\end{array} \) is represented as \([2, 4, 1, 3]\)

• General idea:
  ⊘ Start with partial solution
  ⊘ Nondeterministically select new queen
  ⊘ Check safety of new queen against those already placed
  ⊘ Add new queen to partial solution if compatible; start again with new partial solution
The N Queens Problem (Prolog)

```prolog
queens(N, Qs) :- queens_list(N, Ns), queens(Ns, [], Qs).
queens([], Qs, Qs).
queens(Unplaced, Placed, Qs) :-
    select(Unplaced, Q, NewUnplaced), no_attack(Placed, Q, 1),
    queens(NewUnplaced, [Q|Placed], Qs).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
    Queen =\= Y + Nb, Queen =\= Y - Nb, Nb1 is Nb + 1,
    no_attack(Ys, Queen, Nb1).

select([X|Ys], X, Ys).
select([Y|Ys], X, [Y|Zs]) :- select(Ys, X, Zs).

queens_list(0, []).
queens_list(N, [N|Ns]) :-
    N > 0, N1 is N - 1, queens_list(N1, Ns).
```
The N Queens Problem (CLP(ℜ))

queens(N,Qs) :- constrain_values(N,N,Qs), place_queens(N,Qs).

constrain_values(0, _N, []).  
constrain_values(N, Range, [X|Xs]) :-  
    N > 0, X > 0, X <= Range, N1 = N - 1,  
    constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).

no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-  
    Queen <> Y + Nb,  
    Queen <> Y - Nb,  
    N1 = Nb + 1,  
    no_attack(Ys, Queen, Nb + 1).

place_queens(N, _) :- N = 0.
place_queens(N, Q) :-  
    N > 0,  
    member(N, Q),  
    N1 = N - 1,  
    place_queens(N1, Q).
The N Queens Problem (CLP(ℜ))

- This last program can attack the problem in its most general instance:

```prolog
?- queens(M,N).
N = [], M == 0 ;
M = [1], M == 1 ;
N = [2, 4, 1, 3], M == 4 ;
N = [3, 1, 4, 2], M == 4 ;
N = [5, 2, 4, 1, 3], M == 5 ;
N = [5, 3, 1, 4, 2], M == 5 ;
N = [3, 5, 2, 4, 1], M == 5 ;
N = [2, 5, 3, 1, 4], M == 5
...
```

- Remark: Herbrand terms used to build the data structures

- But also used as constraints (e.g., length of already built list \(Xs\) in `no_attack(Xs, X, 1)`)

- Note that in fact we are using both ℜ and ℱ
The N Queens Problem (CLP(ℜ))
The N Queens Problem (CLP(\(\mathbb{R}\))):

- CLP(\(\mathbb{R}\)) generates internally a set of equations for each board size
- They are non-linear and are thus delayed until instantiation wakes them up

```prolog
?- constrain_values(4, 4, Q).
Q .=. [_t3, _t5, _t13, _t21]
_t3 .<=. 4
_t5 .<=. 4
_t13 .<=. 4
_t21 .<=. 4
0 .<. _t3
0 .<. _t5
0 .<. _t13
0 .<. _t21
0 .<. abs(-_t5 + _t3 - 1)
0 .<. abs(-_t5 + _t3 + 1)
0 .<. abs(-_t13 + _t3 - 2)
0 .<. abs(-_t13 + _t3 + 2)
0 .<. abs(-_t21 + _t3 - 3)
0 .<. abs(-_t21 + _t3 + 3)
0 .<. abs(-_t13 + _t5 - 1)
0 .<. abs(-_t13 + _t5 + 1)
0 .<. abs(-_t21 + _t5 - 2)
0 .<. abs(-_t21 + _t5 + 2)
0 .<. abs(-_t21 + _t13 - 1)
0 .<. abs(-_t21 + _t13 + 1)
```
The N Queens Problem (CLP(\(\Re\)))

- Constraints are (incrementally) simplified as new queens are added:

  ```prolog
  ?- constrain_values(4, 4, Qs), Qs == [3,1|0Qs].
  0Qs == [_t16, _t24]
  Qs == [3, 1, _t16, _t24]
  _t16 <= 4
  _t24 <= 4
  0 < _t16
  0 < _t24
  0 < abs(_t16 + 1)
  0 < abs(_t16 + 5)
  0 < abs(-_t24 + _t16 - 1)
  0 < abs(-_t24 + _t16 + 1)
  0 < abs(-_t16)
  0 < abs(-_t16 + 2)
  0 < abs(-_t24 + 3)
  0 < abs(-_t24 + _t16 - 1)
  0 < abs(-_t24 + _t16 + 1)
  ```

- Bad choices are rejected using constraint consistency:

  ```prolog
  ?- constrain_values(4, 4, Qs), Qs == [3,2|0Qs].
  no
  ```
Finite Domains (I)

- A *finite domain* constraint solver associates each variable with a finite subset of $\mathbb{Z}$
- i.e., $E \in \{-123, -10..4, 10\}$
  (represented as $E :: [-123, -10..4, 10]$ [Eclipse notation] or as $E$ in $\{-123\}\setminus(-10..4)\setminus\{10\}$ [SICStus notation])
- We can:
  - Perform arithmetic operations (+, −, *, /) on the variables
  - Establish linear relationships among arithmetic expressions (# =, # <, # =<)
- Those operations / relationships are intended to narrow the domains of the variables
- **Note:** SICStus requires the use of the `:- use_module(library(clpfd)).` directive in the source code
Finite Domains (II)

Examples:

?- X #= A + B, A in 1..3, B in 3..7.
X in 4..10, A in 1..3, B in 3..7

- The respective minimums and maximums are added
- There is no unique solution

?- X #= A - B, A in 1..3, B in 3..7.
X in -6..0, A in 1..3, B in 3..7

- The min value of X is the min value of A minus the max value of B
- (Similar for the maximum values)

?- X #= A - B, A in 1..3, B in 3..7, X #>= 0.
   A = 3, B = 3, X = 0

- Putting more constraints results in a unique solution.
Finite Domains (III)

Some useful primitives in finite domains:

- `fd_min(X, T)`: the term `T` is the minimum value in the domain of the variable `X`.
  - This can be used to minimize (c.f., maximize) a solution.
  
  ```
  ?- X #= A - B, A in 1..3, B in 3..7, fd_min(X, X).
  A = 1, B = 7, X = -6
  ```

- `domain(Variables, Min, Max)`: A shorthand for several in constraints.

- `labeling(Options, VarList)`: 
  - instantiates variables in `VarList` to values in their domains.
  - `Options` dictates the search order.

  ```
  ?- X*X+Y*Y #= Z*Z, X #>= Y,
     domain([X, Y, Z],1,1000), labeling([], [X,Y,Z]).
  X = 4, Y = 3, Z = 5,
  X = 8, Y = 6, Z = 10,
  X = 12, Y = 5, Z = 13,
  ...
A Project Management Problem (I)

- The job whose dependencies and task lengths are given by: should be finished in 10 time units or less

- Constraints:

```
pn1(A, B, C, D, E, F, G) :-
  A #>= 0, G #=< 10,  
  B #>= A, C #>= A, D #>= A,  
  E #>= B + 1, E #>= C + 2,  
  F #>= C + 2, F #>= D + 3,  
  G #>= E + 4, G #>= F + 1.  
```
A Project Management Problem (II)

- **Query:**
  
  ```prolog
  ?- pn1(A,B,C,D,E,F,G).
  A in 0..4, B in 0..5, C in 0..4,
  D in 0..6, E in 2..6, F in 3..9, G in 6..10.
  ```

- Note the slack of the variables.

- Some additional constraints must be respected as well, but are not shown by default.

- **Minimize the total project time:**
  
  ```prolog
  ?- pn1(A,B,C,D,E,F,G), fd_min(G, G).
      A = 0, B in 0..1, C = 0, D in 0..2,
      E = 2, F in 3..5, G = 6
  ```

- Variables without slack represent critical tasks.
A Project Management Problem (III)

- An alternative setting:

- We can accelerate task $F$ at some cost

$$
\text{pn2}(A, B, C, D, E, F, G, X) :- \\
A \geq 0, \ G \leq 10, \\
B \geq A, \ C \geq A, \ D \geq A, \\
E \geq B + 1, \ E \geq C + 2, \\
F \geq C + 2, \ F \geq D + 3, \\
G \geq E + 4, \ G \geq F + X.
$$

- We do not want to accelerate it more than needed!

$$
? - \ \text{pn2}(A, B, C, D, E, F, G, X), \\
\text{fd_min}(G, G), \ \text{fd_max}(X, X). \\
A = 0, \ B \text{ in } 0..1, \ C = 0, \ D = 0, \\
E = 2, \ F = 3, \ G = 6, \ X = 3.
$$
A Project Management Problem (IV)

- We have two independent tasks B and D whose lengths are not fixed:

- We can finish any of B, D in 2 time units at best

- Some shared resource disallows finishing both tasks in 2 time units: they will take 6 time units
A Project Management Problem (V)

- Constraints describing the net:

```prolog
pn3(A,B,C,D,E,F,G,X,Y) :-
    A #>= 0, G #=< 10,
    X #>= 2, Y #>= 2, X + Y #= 6,
    B #>= A, C #>= A, D #>= A,
    E #>= B + X, E #>= C + 2,
    F #>= C + 2, F #>= D + Y,
    G #>= E + 4, G #>= F + 1.
```

- Query:

```prolog
?- pn3(A,B,C,D,E,F,G,X,Y), fd_min(G,G).
A = 0, B = 0, C = 0, D in 0..1, E = 2,
F in 4..5, X = 2, Y = 4, G = 6
```

- I.e., we must devote more resources to task B
- All tasks but F and D are critical now
- Sometimes, \texttt{fd_min/2} not enough to provide best solution (pending constraints):

```prolog
pn3(A,B,C,D,E,F,G,X,Y),
    labeling([ff, minimize(G)], [A,B,C,D,E,F,G,X,Y]).
```
By far, the fastest implementation

```prolog
queens(N, Qs, Type) :-
    constrain_values(N, N, Qs),
    all_different(Qs), % built-in constraint
    labeling(Type, Qs).
```

```prolog
constrain_values(0, _N, []).
constrain_values(N, Range, [X|Xs]) :-
    N > 0, N1 is N - 1, X in 1 .. Range,
    constrain_values(N1, Range, Xs), no_attack(Xs, X, 1).
```

```prolog
no_attack([], _Queen, _Nb).
no_attack([Y|Ys], Queen, Nb) :-
    Queen \= Y + Nb, Queen \= Y - Nb, Nb1 is Nb + 1,
    no_attack(Ys, Queen, Nb1).
```

Query. `Type` is the type of search desired.

```
?- queens(20, Q, [ff]).
Q = [1,3,5,14,17,4,16,7,12,18,15,19,6,10,20,11,8,2,13,9] ?
```
CLP(\(\mathcal{F}T\)) (a.k.a. Logic Programming)

- Equations over Finite Trees
- Check that two trees are isomorphic (same elements in each level)

\begin{verbatim}
iso(Tree, Tree).
iso(t(R, I1, D1), t(R, I2, D2)) :-
    iso(I1, D2),
    iso(D1, I2).
\end{verbatim}

?- iso(t(a, b, t(X, Y, Z)), t(a, t(u, v, W), L)).
L=b, X=u, Y=v, Z=W ? ;
L=b, X=u, Y=W, Z=v ? ;
L=b, W=t(_C,_B,_A), X=u, Y=t(_C,_A,_B), Z=v ? ;
L=b, W=t(_E,t(_D,_C,_B),_A), X=u, Y=t(_E,_A,t(_D,_B,_C)), Z=v ?
\end{verbatim}
CLP(WE)

- Equations over finite strings
- Primitive constraints: concatenation (.), string length (::)
- Find strings meeting some property:

  ?- "123".z = z."231", z::0.  no
  ?- "123".z = z."231", z::3.  no
  ?- "123".z = z."231", z::1.  z = "1"
  ?- "123".z = z."231", z::4.  z = "1231"
  ?- "123".z = z."231", z::2.  no

- These constraint solvers are very complex
- Often incomplete algorithms are used
CLP((\text{WE},Q))

- Word equations plus arithmetic over \(Q\) (rational numbers)
- Prove that the sequence \(x_{i+2} = |x_{i+1}| - x_i\) has a period of length 9 (for any starting \(x_0, x_1\))
- Strategy: describe the sequence, try to find a subsequence such that the period condition is violated
- Sequence description (syntax is Prolog III slightly modified):

  \begin{verbatim}
  seq(<Y, X>). abs(Y, Y) :- Y >= 0.
  seq(<Y1 - X, Y, X>.U) :-
      seq(<Y, X>.U)
  abs(Y, -Y) :- Y < 0.
  abs(Y, Y1).
  \end{verbatim}

- Query: Is there any 11–element sequence such that the 2–tuple initial seed is different from the 2–tuple final subsequence (the seed of the rest of the sequence)?

  \begin{verbatim}
  ?- seq(U.V.W), U::2, V::7, W::2, U#W.
  fail
  \end{verbatim}
Summarizing

• **In general:**
  ◦ Data structures (Herbrand terms) for free
  ◦ Each logical variable may have constraints associated with it (and with other variables)

• **Problem modeling :**
  ◦ Rules represent the problem at a high level
    * Program structure, modularity
    * Recursion used to set up constraints
  ◦ Constraints encode problem conditions
  ◦ Solutions also expressed as constraints

• **Combinatorial search problems:**
  ◦ CLP languages provide backtracking: enumeration is easy
  ◦ Constraints keep the search space manageable

• **Tackling a problem:**
  ◦ Keep an open mind: often new approaches possible
Complex Constraints

- Some complex constraints allow expressing simpler constraints
- May be operationally treated as passive constraints
- E.g.: cardinality operator \( \#(L, [c_1, \ldots, c_n], U) \) meaning that the number of true constraints lies between \( L \) and \( U \) (which can be variables themselves)
  - If \( L = U = n \), all constraints must hold
  - If \( L = U = 1 \), one and only one constraint must be true
  - Constraining \( U = 0 \), we force the conjunction of the negations to be true
  - Constraining \( L > 0 \), the disjunction of the constraints is specified
- Disjunctive constructive constraint: \( c_1 \lor c_2 \)
  - If properly handled, avoids search and backtracking
  - E.g.: \( nz(X) \leftarrow X > 0 \).
    \[ nz(X) \leftarrow X < 0. \]
    \[ nz(X) \leftarrow X < 0 \lor X > 0. \]
Other Primitives

- CLP(\(\mathcal{X}\)) systems usually provide additional primitives

- E.g.:
  - \texttt{enum(X)} enumerates \(X\) inside its current domain
  - \texttt{maximize(X)} (c.f. \texttt{minimize(X)}) works out maximum (minimum value) for \(X\) under the active constraints
  - \texttt{delay Goal until Condition} specifies when the variables are instantiated enough so that \texttt{Goal} can be effectively executed
    - Its use needs deep knowledge of the constraint system
    - Also widely available in Prolog systems
    - Not really a constraint: control primitive
Implementation Issues: Satisfiability

- Algorithms must be *incremental* in order to be practical
- Incrementality refers to the performance of the algorithm
- It is important that algorithms to decide satisfiability have a good average case behavior
- Common technique: use a *solved form* representation for satisfiable constraints
- Not possible in every domain
- E.g. in $\mathcal{FT}$ constraints are represented in the form $x_1 = t_1(\tilde{y}), \ldots, x_n = t_n(\tilde{y})$, where
  - each $t_i(\tilde{y})$ denotes a term structure containing variables from $\tilde{y}$
  - no variable $x_i$ appears in $\tilde{y}$

(i.e., idempotent substitutions, guaranteed by the unification algorithm)
Implementation Issues: Backtracking in CLP($\mathcal{X}$)

- Implementation of backtracking more complex than in Prolog
- Need to record changes to constraints
- Constraints typically stored as an association of variable to expression
- Trailing expressions is, in general, costly: cannot be stored at every change
- Avoid trailing when there is no choice point between two successive changes
- A standard technique: use *time stamps* to compare the age of the choice point with the age of the variable at the time of last trailing

<table>
<thead>
<tr>
<th>Constraints</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X &lt; Y+Z$, $Y = Z+W$</td>
<td><strong>trail</strong> W, <strong>timestamp it</strong></td>
</tr>
<tr>
<td>$X &lt; Y+4$, $Y = 4+W$, $Z = 4$</td>
<td><strong>trail</strong> X, Y, Z, <strong>timestamp them</strong></td>
</tr>
<tr>
<td>$X &lt; Y+Z$, $Y = Z+W$</td>
<td><strong>timestamp</strong> X, Y, Z, W</td>
</tr>
</tbody>
</table>
Implementation Issues: Extensibility

- Constraint domains often implemented now in Prolog-based systems using:
  - Attributed variables [Neumerkel,Holzbaur]:
    * Provide a hook into unification.
    * Allow attaching an *attribute* to a variable.
    * When unification with that variable occurs, user-defined code is called.
    * Used to implement constraint solvers (and other applications, e.g., distributed execution).
  - Constraint handling rules (CHRs):
    * Higher-level abstraction.
    * Allows defining propagation algorithms (e.g., constraint solvers) in a high-level way.
    * Often translated to attributed variable-based low-level code.
Attributed Variables Example: Freeze

- **Primitives:**
  - attach_attribute(X,C)
  - get_attribute(X,C)
  - detach_attribute(X)
  - update_attribute(X,C)
  - verify_attribute(C,T)
  - combine_attributes(C1,C2)

- **Example: Freeze**

```prolog
freeze( X, Goal) :-
    attach_attribute( V, frozen(V,Goal)),
    X = V.

verify_attribute( frozen(Var,Goal), Value) :-
    detach_attribute( Var),
    Var = Value,
    call(Goal).

combine_attributes( frozen(V1,G1), frozen(V2,G2)) :-
    detach_attribute( V1),
    detach_attribute( V2),
    V1 = V2,
    attach_attribute( V1, frozen(V1,(G1,G2))).
```
Programming Tips

- Over-constraining:
  - Seems to be against general advice “do not perform extra work”, but can actually cut more space search
  - Specially useful if infer is weak
  - Or else, if constraints outside the domain are being used

- Use control primitives (e.g., cut) very sparingly and carefully

- Determinacy is more subtle, (partially due to constraints in non–solved form)

- Choosing a clause does not preclude trying other exclusive clauses (as with Prolog and plain unification)

- Compare:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>max(X,Y,X)</td>
<td>:- X .&gt; Y.</td>
<td>max(X,Y,Z).</td>
</tr>
<tr>
<td>max(X,Y,Y)</td>
<td>:- X .&lt;= Y.</td>
<td>Z =. X, Y .&lt; X</td>
</tr>
</tbody>
</table>

with

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>max(X,Y,X)</td>
<td>:- X .&gt; Y, !.</td>
<td>max(X,Y,Z).</td>
</tr>
<tr>
<td>max(X,Y,Y)</td>
<td>:- X .&lt;= Y.</td>
<td>Z =. X, Y .&lt; X</td>
</tr>
</tbody>
</table>
Some Real Systems (I)

- CLP defines a class of languages obtained by
  - Specifying the particular constraint system(s)
  - Specifying *Computation* and *Selection* rules
- Most share the Herbrand domain with “=”, but add different domains and/or solver algorithms
- Most use *Computation* and *Selection* rules of Prolog

- **CLP(ℜ):**
  - Linear arithmetic over reals (\(=, \leq, >\))
  - Gauss elimination and an adaptation of Simplex

- **PrologIII:**
  - Linear arithmetic over rationals (\(=, \leq, >, \neq\)), Simplex
  - Boolean (\(=\)), 2-valued Boolean Algebra
  - Infinite (rational) trees (\(=, \neq\))
  - Equations over finite strings
Some Real Systems (II)

- **CHIP:**
  - Linear arithmetic over rationals (\(=, \leq, >, \neq\)), Simplex
  - Boolean (\(=\)), larger Boolean algebra (symbolic values)
  - Finite domains
  - User–defined constraints and solver algorithms

- **BNR-Prolog:**
  - Arithmetic over reals (closed intervals) (\(=, \leq, >, \neq\)), Simplex, propagation techniques
  - Boolean (\(=\)), 2-valued Boolean algebra
  - Finite domains, consistency techniques under user–defined strategy

- **SICStus 3:**
  - Linear arithmetic over reals (\(=, \leq, >, \neq\))
  - Linear arithmetic over rationals (\(=, \leq, >, \neq\))
  - Finite domains
Some Real Systems (III)

- **ECLiPSe:**
  - Finite domains
  - Linear arithmetic over reals \((=, \leq, >, \neq)\)
  - Linear arithmetic over rationals \((=, \leq, >, \neq)\)

- **clp(FD)/gprolog:**
  - Finite domains

- **RISC–CLP:**
  - Real arithmetic terms: any arithmetic constraint over reals
  - Improved version of Tarski’s quantifier elimination

- **Ciao:**
  - Linear arithmetic over reals \((=, \leq, >, \neq)\)
  - Linear arithmetic over rationals \((=, \leq, >, \neq)\)
  - Finite Domains

  (can be selected on a per-module basis)

- **Many Prolog systems now support constraints!**