Definitions

- Constraint programming (CP) is a declarative formalism that lets you describe conditions a solution must satisfy.
- CP can be used to model and solve various combinatorial problems such as
  - planning,
  - scheduling
  - allocation of tasks.
CLP in SWI-Prolog

- library(clpfd): Constraint Logic Programming over Finite Domains
- library(clpr): Constraint Logic Programming over Rationals and Reals

1 - library must be loaded explicitly before using it:

```prolog
:- use_module(library(clpq)).
```
Constraint Logic Programming over Finite Domains (clpfd)

- Predicates of clpfd are
  - finite domain constraints, which are relations over integers.
  - generalise arithmetic evaluation of integer expressions in that propagation can proceed in all directions.
- Enumeration predicates let systematically search for solutions on variables whose domains are finite.
Finite domain expressions

- an integer
- a variable
- -Expr
- Expr + Expr
- Expr * Expr
- Expr - Expr
- min(Expr,Expr)
- max(Expr,Expr)
- Expr mod Expr
- abs(Expr)
- Expr / Expr

- Given value
- Unknown value
- Unary minus
- Addition
- Multiplication
- Subtraction
- Minimum of two expressions
- Maximum of two expressions
- Remainder of integer division
- Absolute value
- Integer division
Finite domain constraints

Expr1 $\geq$ Expr2  \(\text{Expr1 is larger than or equal to Expr2}\)
Expr1 $\leq$ Expr2  \(\text{Expr1 is smaller than or equal to Expr2}\)
Expr1 $=$ Expr2  \(\text{Expr1 equals Expr2}\)
Expr1 $\neq$ Expr2  \(\text{Expr1 is not equal to Expr2}\)
Expr1 $>$ Expr2  \(\text{Expr1 is strictly larger than Expr2}\)
Expr1 $<$ Expr2  \(\text{Expr1 is strictly smaller than Expr2}\)

The constraints in/2, #/=2, #\/=2, #</2, #>/2, #=</2, and #>=/2 can be reified, which means reflecting their truth values by integers 0 and 1.
Reifiable constraints and Boolean variables

Let $P$ and $Q$ denote reifiable constraints, then

\[
\begin{align*}
\# \backslash Q & \quad \text{true iff } Q \text{ is false} \\
\# \backslash / Q & \quad \text{true iff } \text{either } P \text{ or } Q \\
\# / \backslash Q & \quad \text{true iff } \text{both } P \text{ and } Q \\
\# <==> Q & \quad \text{true iff } P \text{ and } Q \text{ are equivalent} \\
\# == > Q & \quad \text{true iff } P \text{ implies } Q \\
\# <= = Q & \quad \text{true iff } Q \text{ implies } P
\end{align*}
\]
Example

?- [library(clpfd)].

?- X #> 3.
X in 4..sup.

?- X #\= 20.
X in inf..19 \ 21..sup.

?- 2*X #= 10.
X = 5.

?- X*X #= 144.
X in -12\12.
Example

?- 4*X + 2*Y #= 24, X + Y #= 9, [X,Y] ins 0..sup.
X = 3,
Y = 6.

?- Vs = [X,Y,Z], Vs ins 1..3, all_different(Vs), X = 1, Y \#\= 2.
Vs = [1, 3, 2],
X = 1,
Y = 3,
Z = 2.

?- X #= Y #<===> B, X in 0..3, Y in 4..5.
B = 0,
X in 0..3,
Y in 4..5.
Usage of CLP

- Common scenario:
  1. Post the desired constraints among the variables of a model
  2. use enumeration predicates to search for solutions.

Example of constraint satisfaction problem:
cryptoarithmetic puzzle SEND + MORE = MONEY,
- where different letters denote distinct integers between 0 and 9.
Example (continues)

- Modeling **SEND + MORE = MONEY** in CLP(FD):

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

puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
    Vars = [S,E,N,D,M,O,R,Y],
    Vars ins 0..9,
    all_different(Vars),
    S*1000 + E*100 + N*10 + D +
    M*1000 + O*100 + R*10 + E
    #=
    M*10000 + O*1000 + N*100 + E*10 + Y,
    M #\= 0, S #\= 0. % largest decimal places cannot be 0-s
```
Example (continues)

- Sample query and its result:

?- puzzle(As+Bs=Cs).
As = [9, _G10107, _G10110, _G10113],
Bs = [1, 0, _G10128, _G10107],
Cs = [1, 0, _G10110, _G10107, _G10152],
_G10107 in 4..7,
1000*9+91*_G10107+ -90*_G10110+ _G10113+ -9000*1+ -900*0+10*_G10128+ -1*_G10152#=0,
all_different([_G10107, _G10110, _G10113, _G10128, _G10152, 0, 1, 9]),
_G10110 in 5..8,
_G10113 in 2..8,
_G10128 in 2..8,
_G10152 in 2..8.
Example (continues)

- Constraint solver deduces bounds for all variables.
- Keeping the modeling part separate from the search allows more easily experiment with different search strategies.
- Labeling can then be used to search for solutions:
Example

?- puzzle(As+Bs=Cs), label(As).

As = [9, 5, 6, 7],
Bs = [1, 0, 8, 5],
Cs = [1, 0, 6, 5, 2] ;
false.

% label(As) – is trying out explicit values for the finite domain variables
Variable domain constraints

?Var in +Domain
Var is an element of Domain where the Domain is one of:

- Integer
  Singleton set consisting only of Integer.
- Lower .. Upper
  All integers $I$ such that $\text{Lower} \leq I \leq \text{Upper}$. Lower must be an integer or the atom $\text{inf}$, which denotes negative infinity. Upper must be an integer or the atom $\text{sup}$, which denotes positive infinity.
- Domain1 \/ Domain2
  The union of Domain1 and Domain2.
Variable domain constraints

+Vars ins +Domain
- The variables in the list Vars are elements of Domain.

indomain(?Var)
- Bind Var to all feasible values of its domain on backtracking.
- The domain of Var must be \textit{finite}.
Labeling

labeling(+Options, +Vars)

- Labeling means systematically trying out values for the finite domain variables Vars until all of them are ground.
- The domain of each variable in Vars must be finite.
- +Options is a list of options that exhibits some control over the search process.
- Several categories of options exist
Labeling strategy options

**leftmost** - Label the variables in the order they occur in Vars (that is default)

**ff** - first fail. Label the leftmost variable with smallest domain next, in order to detect infeasibility early. This is often a good strategy.

**ffc** - label the variables with smallest domains, the leftmost one participating in most constraints is labeled next.

**min** - label the leftmost variable next, whose lower bound is the lowest.

**max** - label the leftmost variable next, whose upper bound is the highest.
Labeling strategy options (cont.)

The value order is one of:

**up** - try the elements of the chosen variable's domain in ascending order. This is default.

**down** - try the domain elements in descending order.
Labeling strategy options (cont.)

The branching strategy options:

**step** - for each variable $X$, a choice is made between $X = V$ and $X \neq V$, where $V$ is determined by the value ordering options (default).

**enum** - for each variable $X$, a choice is made between $X = V_1$, $X = V_2$, ..., for all values $V_i$ of the domain of $X$.

The order is determined by the value ordering options.

**bisect** - for each variable $X$, a choice is made between $X \leq M$ and $X > M$, where $M$ is the midpoint of the domain of $X$.

At most one option of each category can be specified, and an option must not occur repeatedly.
The order of solutions option:

\textbf{min}(Expr) - generates solutions in ascending order w.r.t. the evaluation of the arithmetic expression \textit{Expr}

\textbf{max}(Expr) - generates solutions in descending order

- Labeling Vars must make Expr ground.
- If several options are specified, they are interpreted from left to right.
Labeling strategy options (cont.)

- Example:
  
  ?- [X, Y] ins 10..20, labeling([max(X), min(Y)], [X, Y]).

  - This generates solutions of X in descending order,
  - but for each binding of X, solutions of Y are generated in ascending order.
Other labeling options

\texttt{all\_different(+Vars) -}

all variables have pairwise distinct values

\texttt{sum(+Vars, +Rel, ?Expr) -}

The sum of elements of the list \texttt{Vars} is in relation \texttt{Rel} to \texttt{Expr}.

For example:

?- [A,B,C] ins 0..sup, sum([A,B,C], #=, 100).
A in 0..100,
A+B+C#=100,
B in 0..100,
C\_in\_o..100.
Other labeling options

\texttt{scalar\_product(+Cs, +Vs, +Rel, ?Expr)}

- \texttt{Cs} is a list of integer constants,
- \texttt{Vs} is a list of variables and integers.
- True if the scalar product of \texttt{Cs} and \texttt{Vs} is in relation \texttt{Rel} to \texttt{Expr}.

- Example:
  - \texttt{Scalar\_product([4,5], [A,B], >, A-B)}.
  - solves an inequation $4A + 5B > A - B$
Sudoku

sudoku(Rows) :-
   length(Rows, 9), maplist(length_(9), Rows),
   append(Rows, Vs), Vs ins 1..9,
   maplist(all_distinct, Rows),
   transpose(Rows, Columns),
   maplist(all_distinct, Columns),
   Rows = [A,B,C,D,E,F,G,H,I],
   blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).

% maplist(:Goal, ?List) - true if Goal can successfully be applied on all elements of the List.
% maplist(:Goal, ?List₁, ?List₂) - true if Goal can successfully be applied to all successive pairs of elements of List₁ and List₂.
length_(L, Ls) :-
   length(Ls, L).

blocks([], [], []). blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
   all_distinct([A,B,C,D,E,F,G,H,I]),
   blocks(Bs1, Bs2, Bs3).
problem(1,
[[_,_,_,_,_,_,_,_],
[_,_,_,_,_,3,_8,5],
[_,_,1,_,2,_,_,_],
[_,_,_,_,5,_,7,_,_],
[_,_,4,_,_,1,_,_],
[_,9,_,_,_,_,_,_],
[5,_,_,_,_,_,_,7,3],
[_,_,2,_,1,_,_,_],
[_,_,_,4,_,_,_,9]]).
• transpose(+Matrix, ?Transpose).
  
  Transposes a list of lists of the same length.

• Example:

?- transpose([[1,2,3],[4,5,6],[7,8,9]], Ts).
  Ts = [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
?- problem(1, Rows), sudoku(Rows), maplist(writeln, Rows).

[9, 8, 7, 6, 5, 4, 3, 2, 1]
[2, 4, 6, 1, 7, 3, 9, 8, 5]
[3, 5, 1, 9, 2, 8, 7, 4, 6]
[1, 2, 8, 5, 3, 7, 6, 9, 4]
[6, 3, 4, 8, 9, 2, 1, 5, 7]
[7, 9, 5, 4, 6, 1, 8, 3, 2]
[5, 1, 9, 2, 8, 6, 4, 7, 3]
[4, 7, 2, 3, 1, 9, 5, 6, 8]
[8, 6, 3, 7, 4, 5, 2, 1, 9]

Rows = [[[9, 8, 7, 6, 5, 4, 3, 2|...], ..., [...|...]].
Machine games

- **Draughts, English** (Checkers) 8×8 variant of draughts
- weakly solved on April 29, 2007 by the team of Jonathan Schaeffer, known for Chinook,
- Checkers is the largest game that has been solved to date, with a search space of $5 \times 10^{20}$.\[7\]
- The number of calculations involved was $10^{14}$, which were done over a period of 18 years. The process involved 50 - 200 desktop computers