Chapter 4: Presentation of solvers

CONSTRAINT LOGIC PROGRAMMING COURSE

M1 IDO

Overview of CLP Solvers

Constraint Logic Programming (CLP) integrates constraint solving with the logic programming paradigm. Several well-known implementations exist:

GNU-Prolog

Open-source Prolog with a built-in Finite Domain (FD) constraint solver.

ECLiPSe

A powerful CLP platform with rich libraries and constraint handling.

PROLOG IV

Extends Prolog with constraints over reals, rationals, and booleans.

CHIP

One of the earliest CLP systems, pioneering Finite Domain (FD) constraint solving.

The GNU-Prolog Solver

GNU-Prolog is a constraint solver built on the Prolog programming language. It incorporates finite domain constraint solving using arc-consistency-based propagation.

Constraint Types

- Arithmetic constraints
- Boolean constraints
- Symbolic constraints

Predicate Categories

- FD variable declarations
- FD constraints
- Arithmetic predicates
- Boolean predicates
- Symbolic predicates
- Resolution (labeling)

Predicate Description Conventions

Built-in predicates in GNU-Prolog are described using the following conventions:

Designation

predicate_name / arity
e.g. fd_domain/3

Argument Modes

\+ instantiated at call
– output (uninstantiated)
? may or may not be bound

Argument Types

term
variable
Finite Domain variable
list of X

Entry Structure

Call pattern
Description
Example

Finite Domain (FD) Variables

The default domain for any FD variable is [0 .. fd_max_integer]

fd_domain/3 — Range domain

```
fd_domain(+ListOfFDVariables, +Min, +Max)
fd_domain(?FDVariable, +Min, +Max)

% 4-Queens example:
fd_domain([X1,X2,X3,X4], 1, 4)
% or equivalently:
fd_domain(X1,1,4), fd_domain(X2,1,4), fd_domain(X3,1,4), fd_domain(X4,1,4)
```

fd_domain/2 — Explicit value list

```
fd_domain(+ListOfFDVariables, +ListOfIntegers)
fd_domain(?FDVariable, +ListOfIntegers)
% Define domain: X, Y, Z can only be 1, 3, 5, or 7
fd_domain([X, Y, Z], [1, 3, 5, 7])
```

Arithmetic Constraints

Partial Arc Consistency

Relation	Syntax
Equality	`Expr1 #= Expr2`
Inequality	`Expr1 #\= Expr2`
Less than	`Expr1 #< Expr2`
Less than or equal	`Expr1 #=< Expr2`
Greater than	`Expr1 #> Expr2`
Greater or equal	`Expr1 #>= Expr2`

Total Arc Consistency

Relation	Syntax
Equality	`Expr1 #=# Expr2`
Inequality	`Expr1 #\=# Expr2`
Less than	`Expr1 #<# Expr2`
Less than or equal	`Expr1 #=<# Expr2`
Greater than	`Expr1 #># Expr2`
Greater or equal	`Expr1 #>=# Expr2`

Boolean Constraints

Operator	Syntax
Equivalence (biconditional)	`Expr1 #<=> Expr2`
Non-equivalence	`Expr1 #\<=> Expr2`
Negation (NOT)	`#\ Expr1`
Disjunction (OR)	`Expr1 #\/ Expr2`
Exclusive OR (XOR)	`Expr1 ## Expr2`
Conjunction (AND)	`Expr1 #/\ Expr2`
Implication	`#==>`

Examples:

```
(X#=Y) #<=> (Y#=<3)    % X=Y  if and only if  Y <= 3
((X#=Y) #==> (Y#=<3)) #/\ ((X#\=Y) #==> (Y#>3))    % if X=Y then Y<=3, else Y>3
```

Cardinality & Symbolic Constraints

Cardinality Constraints

Restrict how many constraints must be satisfied:

- fd_cardinality/2
- fd_cardinality/3
- fd_at_least_one/1
- fd_at_most_one/1
- fd_only_one/1

Examples:

```
% Exactly 3 of 5 must hold:
fd_cardinality([C1,C2,C3,C4,C5], 3)

% At least 3 must hold:
fd_cardinality([C1..C5], N), N#>=3
```

Symbolic Constraints

fd_all_different/1 — constrains all variables in a list to take pairwise distinct values.

```
fd_all_different(L)
% All vars in L are mutually distinct

% 4-Queens (distinct rows):
fd_all_different([X1,X2,X3,X4])
```

Resolution

fd_labeling/1 — Basic labeling

Assigns values to each FD variable in L to satisfy all constraints.

```
fd_labeling(+ListOfFDVariables)
```

fd_labeling/2 — Labeling with heuristics

fd_labeling(L, O) assigns values to all FD variables in L, guided by the ordering options specified in O.

```
fd_labeling(+ListOfFDVariables, +Options)

% Fail-first heuristic (4-Queens):
fd_labeling([X1,X2,X3,X4], [variable_method(first_fail)])

% Most-constrained variable:
fd_labeling([X1,X2,X3,X4], [variable_method(most_constrained)])
```

General Program Structure

A program for solving a constraint problem follows a three-step structure:

1

Define Domains
Declare the finite domain for each variable using fd_domain/2 or fd_domain/3.

2

Describe Constraints
State the arithmetic, boolean, or symbolic constraints over the variables.

3

Invoke Labeling
Call fd_labeling/1 (or /2) to search for and enumerate solutions.

Example

```
probleme([X,Y,Z]) :- fd_domain(X,0,5),
fd_domain([Y,Z],3,7), X+Y #< 2*Z,
fd_labeling([X,Y,Z],[]).
```

Resolution

```
| ?- probleme(L).
L = [0,3,3] ? ;
L = [0,3,4] ? ;
L = [0,3,5] ? ;
etc.
```

Example: The Knapsack Problem

A knapsack of capacity 9. Select items to achieve a profit of at least 30.

Item	Variable	Profit	Weight
Juice	J	15	4
Perfume	P	10	3
Cigarettes	C	7	2

Constraints:

```
4*J + 3*P + 2*C   =<   9          (capacity constraint)
15*J + 10*P + 7*C  >=  30         (profit constraint)
```

Knapsack Problem — Solutions

GNU-Prolog

```
sac_dos([J,P,C]) :-
  fd_domain([J,P,C], 0, 9),
  4*J+3*P+2*C #=< 9,
  15*J+10*P+7*C #>= 30,
  fd_labeling([J,P,C]).

% Solutions:
J=0, P=1, C=3
J=0, P=3, C=0
J=1, P=1, C=1
J=2, P=0, C=0
```

SWI-Prolog

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

sac_dos([J,P,C]) :-
  [J,P,C] ins 0..9,
  4*J+3*P+2*C #=< 9,
  15*J+10*P+7*C #>= 30,
  label([J,P,C]).

% Solutions:
J=0, P=1, C=3
J=0, P=3, C=0
J=P=C=1
```