On Teaching Constraint-based Modeling and Algorithms for Decision Support in Prolog

François Fages[1]

[1]*Inria Saclay and Ecole Polytechnique, Palaiseau, France*

Abstract
Constraint programming techniques are particularly successful at solving discrete optimization problems such as resource allocation, scheduling or transport problems which are ubiquitous in the industry. Although historically introduced in the mid 80's with the generalization of Prolog to the class of Constraint Logic Programming languages, those techniques are mainly used today through constraint solving libraries in standard programming languages such as C++, Java, Python, and mainly taught with constraint-based modeling languages such as MiniZinc or Essence, in the tradition of algebraic modeling languages developed for mixed integer linear programming. Nonetheless, the foundations of both constraint solvers and constraint-based models in first-order logic should make of Prolog with its constraint-solving libraries a unique language to teach all aspects of constraint programming, provided missing higher-level MiniZinc-like mathematical modeling language constructs are added to Prolog libraries. This is what I have developed for teaching purposes in SWI-Prolog through a package named modeling. This package contains libraries for defining shorthand functional notations, subscripted variables (arrays in addition to lists), set comprehension (bounded iteration and quantifiers compatible with constraints, in addition to recursion), front-end to constraint solving libraries for shorthand expansions on arrays and reification, and search tree tracing and visualization. This approach makes it possible to teach constraint-based modeling, search programming, and constraint solving with a unique high-level modeling/programming language, Prolog.

Keywords
constraint programming, combinatorial optimization, algebraic modeling languages, MiniZinc, meta-predicates, set comprehension, answer constraint semantics, constraint solving

1. Introduction

The problem of placing N queens on a chessboard of size $N \times N$ such that they do not attack each other (i.e. not placed on the same column, row or diagonal) is classically modeled and solved in Prolog with a finite domain constraint solver in the framework of Constraint Logic Programming (CLP) [1], by

- creating a list of N finite domain decision variables, representing say the queens in each column, with domain $1..N$, representing the rows where they are placed,
- and posting disequality constraints between each pair of queens, by double recursion on the list of variables and their tail list.

However, for a course on constraint-based modeling and solving of combinatorial problems in an engineering school, it is more natural to use mathematical notations on subscripted variables,

and set comprehension for posting constraints by iteration on indices rather than by double recursion on lists. This is more in the spirit of mathematical modeling languages, such as algebraic modeling languages developed for mixed integer linear programming, or modeling languages like MiniZinc [2, 3] or Essence [4] for constraint programming.

For example, one can compare the following MiniZinc-like model using our SWI-Prolog package named `modeling`[1] [5], with the classical CLP program to solve the N-queens problem:

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

queens(N, Queens):-
  int_array(Queens, [N], 1..N),
  for_all([I in 1..N-1, D in 1..N-I],
          (Queens[I] #\= Queens[I+D],
           Queens[I] #\= Queens[I+D]+D,
           Queens[I] #\= Queens[I+D]-D)),
  satisfy(Queens).


?- queens(N, Queens).
N = 1,
Queens = array(1) ;
N = 4,
Queens = array(2, 4, 1, 3) ;
N = 4,
Queens = array(3, 1, 4, 2) ;
N = 5,
Queens = array(1, 3, 5, 2, 4) .
```

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

queens(N, Queens):-
  length(Queens, N),
  Queens ins 1..N,
  safe(Queens),
  label(Queens).

safe([]).
safe([QI | Tail]) :-
  noattack(Tail, QI, 1),
  safe(Tail).

noattack([], _, _).
noattack([QJ | Tail], QI, D):-
  QI #\= QJ,
  QI #\= QJ + D,
  QI #\= QJ - D,
  D1 #= D + 1,
  noattack(Tail, QI, D1).
```

The writing of the model using subscripted variables and set comprehension is arguably higher-level, since it does not involve recursion and termination proof, but simply bounded quantification on pairs of indices defined by set comprehension. This is even more striking when it comes to breaking all symetries of the chessboard square, i.e. of the dihedral group of order 8 including reflection symetries around the vertical, horizontal, diagonal, antidiagonal axes and rotation symetries, using lexicographic ordering on the primal and dual models expressed by playing on the indices, rather than by tricky list recursions [5].

MiniZinc was designed with a principle of independence of constraint solvers, in order to compare their performances on common benchmarks of models. A MiniZinc model is usually transformed in a FlatZinc model in which the high-level constructs have been eliminated and replaced by a flat constraint satisfaction problem that can be executed by a variety of constraint solvers parsing FlatZinc syntax. This is the way for instance SICStus-Prolog is interfaced with FlatZinc as a general purpose constraint solver to solve problems modeled in MiniZinc [6].

For teaching purposes however, we are more interested in having a MiniZinc interpreter in Prolog in order to combine pure constraint-based modeling aspects with programming aspects for programming search or data interface. This is possible in Eclipse system [7] but, to the best of our knowledge, did not exist in the form of Prolog libraries. This was thus our main motivation
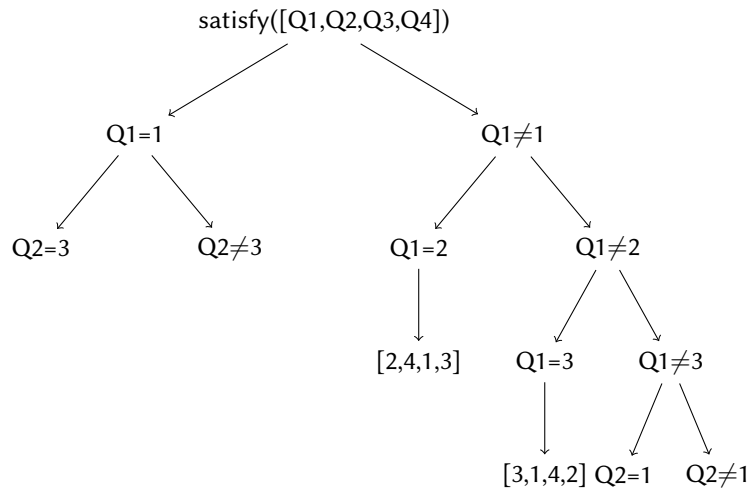
---

[1]https://eu.swi-prolog.org/pack/list?p=modeling

Figure 1: Search tree term automatically generated by trace option for enumerating all solutions to the 4 queens problem. The students can see the active use of constraints which have for effect to decrease the domains of the variables and prune the search tree in advance during search, in sharp contrast to a simple "generate and test" program by backtracking, that would generate the 4^4 possible valuations of the variables to find the 2 solutions.

for developing a `modeling` pack to introduce functional notations and set comprehension predicates, and write constraint-based mathematical models *à la* MiniZinc in Prolog.

Making these high-level metapredicates available in Prolog for the pure modeling part, makes it possible to combine them with full programming features of Prolog for data interfaces, programming search, representing the search tree by a Prolog term, and visualizing it, a very important feature for teaching constraint programming. Fig. 1 displays the search tree term generated by `satisfy/1` predicate with trace option of our modeling library for computing all solutions to `queens(4, Q)` query.

The possibility of combining constraint domains and reifying constraints with Boolean variables, also makes of CLP a very expressive modeling language, compared to more traditional algebraic modeling languages developed for linear programming. For example, the computation of magic series $(x_1, \ldots, x_n)$ satisfying the equation

$$\forall\, i \in \{1, \ldots, n\} \ \ x_i \; = \; \mathrm{card}\, \{j \in \{1, \ldots n\} \mid x_j = i - 1\}$$

i.e. series where x_i is the number of occurrences of the value $i - 1$ in the series, can be solved by a quite direct transcription of the mathematical definition of the problem as follows:

```
magic_series(N, X):-
    array(X, [N]),
    for_all(I in 1..N,
            X[I] #= int_sum(J in 1..N,
                            truth_value(X[J] #= I-1))),
    satisfy(X).
```

```
?- magic_series(N, X).
X = array(1, 2, 1, 0),
N = 4 ;
X = array(2, 0, 2, 0),
N = 4 ;
X = array(2, 1, 2, 0, 0),
N = 5 ;
X = array(3, 2, 1, 1, 0, 0, 0),
N = 7 ;
X = array(4, 2, 1, 0, 1, 0, 0, 0),
N = 8 .
X = array(5, 2, 1, 0, 0, 1, 0, 0, 0),
N = 9 .
```

In this definition, the Boolean truth values in $\{0, 1\}$ of all constraints $x_j = i - 1$ for $1 \leq j \leq n$ are summed as integers, and the sum is constrained to be equal to x_i. That sum is expressed using general-purpose shorthand functional notations introduced for predicates where one argument can be interpreted as a result, here the last argument for constraint int_sum/3 and reification constraint truth_value/2.

These features added to Prolog with libraries allow us to teach constraint programming techniques in a unique programming environment, by focusing successively

- on constraint-based mathematical modeling techniques for a variety of constraint satisfaction problems at a high-level of abstraction,
- on general purpose constraint solvers programmed in Prolog for different domains,
- and on search procedures including heuristic search and constraint-based local search procedures written as well in Prolog.

2. Shorthand functional notations

Mathematical notations necessitate allowing shorthand functional notations in Prolog expressions and goals. We have defined in library(shorthand) some general purpose metapredicates shorthand/3, expand/2, expand/1, to respectively define new shorthand notations, and expanding them in a term or in a goal before executing it.

For example, this is the way the Array[Indices] functional notation for subscripted variables is defined in SWI-Prolog in library(array), as a shorthand for V satisfying the predicate cell(Array, Indices, V), by:

```
:- op(100, yf, []).
user:shorthand([](Indices, Array), V, cell(Array, Indices, V)) :- !.
```

Independently of the particular syntax used in this example, shorthand functional notations can be defined for any predicate in which one argument can be interpreted as a result, e.g. for cell/2 functional notation without using a particular syntax, with respect to the last argument of cell/3 predicate, by:

```
user:shorthand(cell(Array, Indices), V, cell(Array, Indices, V)) :- !.
```

3. Comprehension metapredicates with answer constraint semantics

Most aggregation metapredicates in Prolog are based on a special mechanism introduced by David Warren in [8] to collect information across backtracking, and illustrated in his seminal paper with the `setof(X, P, S)` metapredicate, for collecting in S the set of instances of X such that P is provable. In this approach, a non deterministic goal P is thus used as generator comprehension by backtracking. This mechanism used for higher-order programming in Prolog [9], and quantifiers like `forall/3`, cannot be used to constrain context variables, since they refer to the success semantics of Prolog, not the answer constraint semantics [1].

We had thus to introduce in `library(comprehension)` a few metapredicates for generator comprehension by iteration, rather than by backtracking, in order to satisfy the logical semantics of constraints. For example, we introduced a universal quantifier `for_all/3` capable of constraing context variables to satisfy the goal for all instances satisfying the condition, similarly to lower-level `maplist/2` which requires specifying context and lambda variables explicitly (operational view), but in sharp contrast to ISO Prolog `forall/3` metapredicate which merely tests goal satisfiability on all elements:

```
?- L=[A, B, C], forall(member(X, L), 1=X). % satisfiability test for all elements
L = [A, B, C].

?- L=[A, B, C], for_all(X in L, 1=X). % constraint posted for all elements
L = [1, 1, 1],
A = B, B = C, C = 1.

?- L=[A,B,C], maplist([X]>>(1=X), L). % lambda constraint posted for all elements
L = [1, 1, 1],
A = B, B = C, C = 1.

?- forall(member(X, [1, 2, 3]), X #< Y).
true.

?- for_all(X in [1, 2, 3], X #< Y).
clpfd:(Y in 4..sup).

?- maplist({Y}/[X]>>(X #< Y), [1, 2, 3]).
clpfd:(Y in 4..sup).
```

We similarly found it useful to introduce a comprehension metapredicate `aggregate_for/6` of hopefuly easier use for the students than `foldl/n`.

4. Front-end to constraint solving libraries

We have defined `library(clp)` as a front-end to `clpfd` and `clpr` constraint solving libraries on, respectively, $(\mathbb{Z}, +, *, ..., \#=, \#>, ...)$ and $(\mathbb{R}, +, *, ..., \{=\}, \{>\}, ...)$. This allows us also to expand shorthand notations in constraints, accept indifferently arrays or lists in the

arguments of global constraints, reify `clpr` constraints with `clpfd` Boolean variables, and define hybrid constraints.

Furthermore, the `labeling/2` predicate is enriched in this front-end library with a new option for tracing the search tree and visualizing it in different forms (e.g. generating LaTeX tikz used for Fig. 1).

5. Modeling library

Finally, `library(modeling)` defines predicates for creating arrays of Boolean, integer or floating point decision variables, specify their domains using shorthand notations, and solve constraint satisfaction and optimization problems with default search strategies.

6. Plan of the course

With this modeling package in Prolog, it is possible to give a course on "Constraint-based modeling and algorithms for decision support in Prolog" at Master level, and in part at Bachelor 3rd year level,

- by focusing first on constraint-based mathematical modeling techniques for solving various combinatorial problems, through a simple use of the modeling library,
- before getting into the foundations in first-order logic of constraint languages and logic programming,
- the programming of general purpose constraint solvers,
- and the study of search procedures.

One possible plan for the course in this approach is as follows:

1. Solving constraint satisfaction problems with general purpose constraint solvers
   a) Problem modeling with decision variables, domains and constraints
   b) Solving logical and arithmetical puzzles
   c) Solving production planning problems
2. Constraint logic programs
   a) Deductive databases in Datalog
   b) First-order logic terms, unification algorithm, metaprogramming
   c) List processing, task scheduling
   d) Operational semantics of CLP(X) languages and logical semantics
3. Constraint-based modeling and solvers over $\mathbb{R}$
   a) Fourier's elimination algorithm
   b) Symbolic answer constraints
   c) Dantzig's simplex algorithm, Farkas's duality
   d) Production planning and cost-benefit analysis
4. Constraint-based modeling and solvers over $\mathbb{Z}$
   a) Domain filtering algorithms
   b) Reified constraints

    c) Constraints as closure operators in a domain lattice
    d) Resource allocation and disjunctive scheduling problems

5. Search heuristic
    a) Search procedures and meta-interpreters
    b) Automated theorem proving
    c) And-choice heuristics
    d) Or-choice heuristics
    e) AI planning

6. Symmetry breaking constraints
    a) Variable symmetries
    b) Value symmetries
    c) Valuation symmetries
    d) Symmetry breaking during search

7. Boolean modeling and SAT solvers
    a) Boolean models
    b) Computational complexity classes
    c) DPLL algorithm
    d) Conflict driven clause learning
    e) Phase transition in random SAT

8. Constraint-based local search algorithms
    a) Constraint violation functions
    b) Greedy hill climbing
    c) Random walk
    d) Simulated annealing
    e) Tabu search

9. Final comments on effective modeling practices
    a) Choice of decision variables
    b) Choice of constraints and their generation
    c) Choice of search procedure
    d) Cooperation of constraint solvers
    e) Creation of global constraints
    f) Learning constraint-based models from examples

In such a course, Prolog with modeling pack provides an end-to-end solution to cover the different aspects of constraint programming techniques for knowledge representation and combinatorial optimisation, going from constraint-based modeling to relation-based programming, with their common foundations in first-order logic.

Acknowledgments

References

[1] J. Jaffar, J.-L. Lassez, Constraint logic programming, in: Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany, ACM, 1987, pp. 111–119.

[2] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: Proc. 13th Int. Conf. on Principles and Practice of Constraint Programming, CP'07, volume 4741 of *LNCS*, Springer-Verlag, 2007, pp. 529–543.

[3] The Zinc team, MiniZinc web page, 2023.
`http://www.minizinc.org/`.

[4] A. M. Frisch, W. Harvey, C. Jefferson, B. Martinez-Hernandez, I. Miguel, Essence: A constraint language for specifying combinatorial problems, Constraints 13 (2008) 268–306.

[5] F. Fages, A constraint-based mathematical modeling library in prolog with answer constraint semantics, in: 17th International Symposium on Functional and Logic Programming, FLOPS 2024, volume 14659 of *LNCS*, Springer-Verlag, Kumamoto, Japan, 2024, pp. 135–150. URL: https://inria.hal.science/hal-04478132. doi:`10.1007/978-981-97-2300-3_8`.

[6] Mats Carlsson et al., Sicstus 4.2.3, 2012.
`https://sicstus.sics.se/`.

[7] K. Apt, M. Wallace, Constraint Logic Programming using Eclipse, Cambridge University Press, 2006.

[8] D. H. D. Warren, Higher-order extensions to Prolog: Are they needed?, in: Machine Intelligence, volume 10, 1982, pp. 441–454.

[9] K. Sagonas, D. S. Warren, Effcient execution of HiLog in WAM-based prolog implementations, in: L. Sterling (Ed.), Proceedings of the 12th International Conference on Logic Programming, MIT Press, 1995, pp. 349–363.