7F−2

Finite-domain Constraint Programming in Beta-Prolog

周 能法* 長沢 勲†

九州工業大学‡

1 Introduction

Introducing finite-domain constraint solving techniques into Prolog is one of the most important extensions of Prolog. It enables the programmer to describe a variety of combinatorial problems declaratively and the system to solve them efficiently [3, 4]. There are two main approaches to introducing finite-domain constraint solving techniques into Prolog. One is adopted in the CHIP system [3, 1], which extends the unification operation to handle domain variables and the computation rule of Prolog to support coroutining. The other approach is to implement finite-domain constraint solver on top of Prolog [2]. Compared with the former approach, the latter is simple. It does not need complicated abstract machines and compilers, and generally do not cause any overhead to the Prolog system. However, it is not efficient enough because Prolog does not provide efficient data structures for representing and handling domains, domain variables, and constraints.

Beta-Prolog is an extended Prolog that provides the following two new facilities: (1) the definition and manipulation of Boolean tables, and (2) constant time update of the arguments of compound terms. In this paper, we describe an efficient finite-domain constraint solver on top of Beta-Prolog. For a constraint satisfaction problem (CSP), domains of variables are represented as a Boolean table, domain variables are represented as compound terms, and constraints are represented as calls in the form c(Constr). Due to the new facilities of Beta-Prolog, the system can solve CSPs quite efficiently. For many benchmark programs, its performance is comparable with that of the CHIP system.

2 Beta-Prolog

This section describes briefly the new facilities of Beta-Prolog and their implementation.

Boolean Table

A Boolean table is a relation whose tuples are associated with a state of either true or false. A Boolean table or a part of a Boolean table is declared as follows:

:- bt(p($X_1, X_2, \ldots, X_n$),S).

where p is the name of the table, each $X_i (1 \le i \le n)$ is a *set expression*, and S is either *true* or *false* which denotes the state of the tuples. This declaration says that the Cartesian product $X_1 \times X_2 \times \ldots \times X_n$ is a part in the Boolean table named p. A set expression is a variable, a list of atomic terms, or a range between two integers.

*Neng-Fa ZHOU
†Isao NAGASAWA
‡Kyushu Institute of Technology

The following primitives on Boolean tables are provided:

(1) select(p($+X_1, \ldots, +X_k, -X_{k+1}, \ldots, -X_n$))
(2) next(p($+X_1, \ldots, +X_n$),$-Y$)
(3) true(p($+X_1, +X_2, \ldots, +X_n$))
(4) false(p($+X_1, +X_2, \ldots, +X_n$))
(5) set_true(p($+X_1, +X_2, \ldots, +X_n$))
(6) set_false(p($+X_1, +X_2, \ldots, +X_n$))
(7) set_false0(p($+X_1, +X_2, \ldots, +X_n$))
(8) count(p($+X_1, \ldots, +X_k$,_,...,_),N)
(9) max(p($+X_1, \ldots, +X_k, -X_{k+1}$))
(10) min(p($+X_1, \ldots, +X_k, -X_{k+1}$))

These primitives have respectively the following meanings: (1) selects a true tuple from p, (2) gets the next true tuple following a given true tuple, (3) and (4) test the state of a given tuple, (5), (6) and (7) set the state of a given tuple, (8) counts the number of true tuples in a block, (9) and (10) select a true tuple whose $k+1$th argument is respectively the maximum and the minimum.

A Boolean table is organized in such a way that all the primitives on it can be performed in constant time [5]. In order to avoid causing any overhead to usual Prolog programs, we introduced two new stacks into the TOAM [6].

Destructive Update

Besides the primitives on compound terms provided by Prolog, the following primitives are introduced:

(1) set_arg(+N,+T,+A)
(2) increment_arg(+N,+T)
(3) decrement_arg(+N,+T)

(1) sets the Nth argument of T to be A, (2) increments the Nth argument of T by one, and (3) decrements the Nth argument of T by one. The updates are undone upon backtracking.

3 Specification and Interpretation of Constraints

This section first describes how CSPs are specified and then describes an interpreter for solving CSPs in Beta-Prolog.

Specification of CSPs

For a CSP, each variable is represented by a Prolog term in the following form:

dvar(ID,Value,Count,Cs),

where ID is the identifier (an atomic value) of the variable, $Value$ is a logical variable that will hold the value to

$$\begin{array}{c} X_7\ X_8\ X_9\ X_{10} \\ X_4\ X_5\ X_6 \\ X_2\ X_3 \\ X_1 \end{array}$$

Figure 1: An arithmetic puzzle.

be assigned to the domain variable, *Cs* is the list of constraints related to the variable and *Count* is the number of constraints in *Cs*.

The domains of variables are represented by a binary Boolean table named domain. For each variable whose identifier is *ID* and each element *E* in its domain, there is a tuple (*ID*,*E*) in the table. Initially, all tuples are true. The select mode of domain is defaultly assumed to be domain(+,-).

Each constraint is represented as a call in the following form:

$$c(Exp_1\ R\ Exp_2)$$

where Exp_1 and Exp_2 are linear expressions and *R* denotes one of the following relation symbols: $=$, $\neq$, $>$, $\geq$, $<$, and $\leq$.

Besides specifying the variables, domains, and constraints, the programmer also need to define a predicate called labeling(Vars), which specifies the order in which variables are processed. Before considering how constraints are handled, we give an example illustrating how CSPs are specified.

Example

Given ten variables as shown in Figure 1, we already know that X_1 is equal to 3, we want to give each variable a different integer from {1,...,10} so that for any three variables in the form

$$\begin{array}{c} X_i\ X_j \\ X_k \end{array}$$

$X_i - X_j = X_k$ or $X_j - X_i = X_k$.

The program shown in Figure 2 specifies the problem. The predicate generate_puzzle(Vars) generates the domain variables and the constraints on them. The predicate write_values(Vars) writes the *Value* components of the variables in *Vars*. The predicate number_vars(Vars,N) translates the variables in *Vars* into internal representation and numbers them with N, $N+1$, and so on. The predicate alldifferent(Vars) generates the constraints that all variables in *Vars* will get different values. The predicate xyz_or_yxz(X,Y,Z) represents the or constraint among three variables.

Interpretation of Constraints

When the call c(Constr) is executed, *Constr* is first transformed into a canonical form. If the canonical form contains no more than one domain variable, then it is solved immediately; otherwise, it is inserted to the lists of related constraints of all the domain variables. When a domain variable is assigned a value, the proagation procedure is invoked to to see whether or not the related constraints are solvable.

```
:- bt(domain(1..10,1..10),true).
solve_puzzle:-
    generate_puzzle(Vars),
    labeling(Vars),
    write_values(Vars).
generate_puzzle(Vars):-
    Vars=[X1,X2,X3,X4,X5,X6,X7,X8,X9,X10],
    number_vars(Vars,1),
    alldifferent(Vars),
    c(X1=3),
    xyz_or_yxz(X2,X3,X1),
    xyz_or_yxz(X4,X5,X2),
    xyz_or_yxz(X5,X6,X3),
    xyz_or_yxz(X7,X8,X4),
    xyz_or_yxz(X8,X9,X5),
    xyz_or_yxz(X9,X10,X6).
xyz_or_yxz(X,Y,Z):-
    c(X-Y=Z).
xyz_or_yxz(X,Y,Z):-
    c(Y-X=Z).
```

Figure 2: A program that specifies the arithmetic puzzle.

Problem	CHIP (VAX-785)	Beta (SPARC-2)
queens96	36.23	3.40
color110	5.55	0.20
sendmory	0.08	0.05
five-houses	1.49	0.10

Table 1: Comparison of execution time (seconds).

4 Conclusion

In this paper, we described a simple but efficient method for introducing finite-domain constraint solving techniques into Prolog. We first described Beta-Prolog, our extended Prolog that supports Boolean tables and provides several new primitives. We then presented a constraint solver on top of Beta-Prolog. The constraint solver is simple. It does not even use the delay primitive. The Prolog system does not cause any overhead when executing usual Prolog programs. It is also quite efficient. For several benchmark problems, its efficiency is comparable with that of the CHIP system (see Table 1).

References

[1] A. Aggoun and N. Beldiceanu : Overview of the CHIP Compiler System, Proc. of the 8th ICLP, Ed., K. Furukawa, pp.775-789, 1991.

[2] D. De Schreye, D. Pollet, J. Ronsyn, M. Bruynooghe: Implementing finite-domain constraint logic programming on top of a Prolog-system with delay-mechanism, Report CW-104, K.U.Leuven, 1989.

[3] P. Van Hentenryck: Constraint Satisfaction in Logic Programming, The MIT Press, 1989.

[4] P. Van Hentenryck, H. Simonis and M. Dincbas: Constraint satisfaction using constraint logic programming, Artif. Intell., 59, pp.113-159, 1992.

[5] N.F. Zhou: Constant time select, test, and update for Boolean tables, Proc. 9th National Conference on Software Technology, JSSST, Keio University, pp.353-356, 1992.

[6] N.F. Zhou: Global Optimizations in a Prolog Compiler for the TOAM, to appear in J. Logic Programming, 1993.