Centrum voor Wiskunde en Informatica
REPORTRAPPORT

Arrays, bounded quantification and iteration in logic and constraint logic programming

K.R. Apt

Computer Science/Department of Software Technology

CS-R9543 1995

Report CS-R9543
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Arrays, Bounded Quantification and Iteration in Logic and Constraint Logic Programming

Krzysztof R. Apt
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
and
Department of Mathematics and Computer Science
University of Amsterdam, Plantage Muidergracht 24
1018 TV Amsterdam, The Netherlands

Abstract

We claim that programming within the logic programming paradigm suffers from lack of attention given to iteration and arrays. To convince the reader about their merits we present several examples of logic and constraint logic programs which use iteration and arrays instead of explicit recursion and lists. These programs are substantially simpler than their counterparts written in the conventional way. They are easier to write and to understand, are guaranteed to terminate and their declarative character makes it simpler to argue about their correctness. Iteration is implemented by means of bounded quantification.

Note. This paper will appear as an invited lecture in: Proc. of the 1995 Joint Conference on Declarative Programming (GULP-PRODE '95), M. Alpuente Frasnedo and M. I. Sessa, editors, University of Salerno, Italy.

AMS Subject Classification (1991): 68Q40, 68T15.
CR Subject Classification (1991): D.1.6, D.3.3, F.3.3, I.2.3.
Keywords and Phrases: arrays, bounded quantifiers, iteration, logic programming, constraints, termination.

1 Introduction

Any systematic course on programming in the imperative style (say using Pascal), first concentrates on iteration constructs (say **while** or **repeat**) and only later deals with recursion. Further, the data structures are explained first by dealing with the static data structures (like arrays and records) and only later with the dynamic data structures (which are constructed by means of pointers).

In the logic programming framework the distinctions between iteration and recursion, and between static and dynamic data structures are lost. One shows that recursion is powerful enough to simulate iteration and rediscovers the latter by identifying it with tail recursion. Arrays do not exist. In contrast, records can be modelled by terms, and dynamic data structures can be defined by means of clauses, in a recursive fashion (with the exception of lists for which in Prolog there is support in the form of built-ins and a more friendly notation).

One of the side effects of this approach to programming is that one often uses a sledgehammer to cut the top of an egg. Even worse, simple problems have unnecessarily complex and clumsy solutions in which recursion is used when a much easier solution using iteration exists, is simpler to write and understand, and — perhaps even more important — is closer to the original specification.

In this paper we would like to propose an alternative approach to programming in logic programming and in constraint logic programming — an approach in which adequate stress is put on the use of arrays and iteration. Because iteration can be expressed by means of bounded quantification, a purely logical construct, the logic programming paradigm is not "violated". On the contrary, it is enriched, clarified and better tailored for the programming needs.

Arrays are especially natural when dealing with vectors and matrices. The use of dynamic data structures to write programs dealing with such objects is unnatural. We shall try to illustrate this point by presenting particularly simple solutions to problems such as the n-queens problem, the knight's tour, the map colouring problem, the cutting stock problem, and other problems involving backtracking.

Further, by adding to the language operators which allow us to express optimization, i.e. minimization and maximization, we can easily write programs for various optimization problems.

For pedagogical reasons we limit our attention to programs that involve iteration and optimization constructs. Of course, explicit recursion has its place both in logic programming and in constraint logic programming. One of the main purposes of this paper is to illustrate how much can be achieved without it.

In the programs considered in this paper recursion is hidden in the implementation of the bounded quanfiers and this use of recursion is guaranteed to terminate. Consequently, these programs always terminate. As termination is one of the major concerns in the case of logic programming, from the correctness point of view it is better to use iteration instead of recursion, when a choice arises. Also, iteration can be implemented more efficiently than recursion (see Barklund and Bevemyr [BB93] for an explanation how to extend WAM to implement iteration in Prolog).

This work has a preliminary character and can be seen as an attempt to identify the right linguistic concepts which simplify programming in the logic programming paradigm. When presenting this view of programming within the logic programming paradigm we were very much influenced by the publications of Barklund and Millroth [BM94], Voronkov [Vor92] and Kluźniak [Klu93]. In fact, the constructs whose use we advocate, i.e. bounded quantification and arrays, were already proposed in these papers. The only, possibly new, contribution of this paper is a suggestion to include these constructs in constraint logic programming.

2 Bounded Quantifiers

Bounded quantifiers in logic programming were introduced in Kluźniak [Klu91] and are thoroughly discussed in Voronkov [Vor92] (where also earlier references in Russian are given). They are also used in Kluźniak [Klu93] (see also Kluźniak and Miłkowska [KM94]) in a specification language SPILL-2 in which executable specifications can be written in the logic programming style.

Following Voronkov [Vor92] we write them as $\exists \mathtt{X} \in \mathtt{L}\ \mathtt{Q}$ (the bounded existential quantifier) and $\forall \mathtt{X} \in \mathtt{L}\ \mathtt{Q}$ (the bounded universal quantifier), where `L` is a list and `Q` a query, and define them as follows:

```
∃X ∈ [Y | Ys] Q ← Q{X/Y}.
∃X ∈ [Y | Ys] Q ← ∃X ∈ Ys Q.

∀X ∈ [Y | Ys] Q ← Q{X/Y}, ∀X ∈ Ys Q.
∀X ∈ [] Q.
```

Voronkov [Vor92] also discusses two other bounded quantifiers, written as ∃X ⊏ L Q and ∀X ⊏ L Q, where X ⊏ L is to be read "X is a suffix of L", which we do not consider here.

To some extent the use of bounded quantifiers allows us to introduce in some compact form the "and" and the "or" branching within the program computations. This reveals some connections with the approach of Harel [Har80], though we believe that the expressiveness and ease of programming within the logic programming paradigm makes Harel's programming proposal obsolete.

Even without the use of arrays the gain in expressiveness achieved by means of bounded quantifiers is quite spectacular. Consider for example the following problem.

Problem 1 Write a program which tests whether one list is a subset of another.

Solution

```
subset(Xs, Ys) ← ∀X ∈ Xs ∃Y ∈ Ys X = Y.
```

Several other examples can be found in Voronkov [Vor92]. Here we content ourselves with just one more, in which we use delay declarations very much like in modern versions of Prolog, (for example in $\mathrm{ECL}^i\mathrm{PS}^e$) or the programming language Gödel of Hill and Lloyd [HL94]).

Problem 2 Write a program checking the satisfiability of a Boolean formula.

Solution We assume here that the input Boolean formula is written using Prolog notation, so for example (¬ X, Y) ; Z stands for (¬ X ∧ Y) ∨ Z.

```
sat(X) ← X, generate(X).
generate(X) ← vars(X, Ls), ∀Y ∈ Ls ∃Z ∈ [true, fail] Y = Z.

DELAY X UNTIL nonvar(X).
```

Comments This remarkably short program uses meta-variables and a mild extension of the delay declarations to meta-variables. The delay declaration used here delays any call to a meta-variable until it becomes instantiated. `vars(t, Ls)` for a term `t` computes in `Ls` the list of the variables occurring in `t`. Its definition is omitted. `vars(X, Ls)` can be easily implemented using the `var(X)` and `univ` built-in's of Prolog. `true` and `fail` are Prolog's built-in's.

In this program it is not advisable to delay the calls to negative literals until they become ground. Such a delay would reduce checking for satisfiability of subformulas which begin with the negation sign to a naive generate and test method.

Even though this program shows the power of Prolog, we prefer to take another course and use types instead of exploring extensions of Prolog, which is an untyped language.

3 Arrays and Bounded Quantifiers in Logic Programming

Arrays in logic programming were introduced in Eriksson and Rayner [ER84]. Barklund and Bevemyr [BB93] proposed to extend Prolog with arrays and studied their use in conjunction with the bounded quantification. In our opinion the resulting extension (unavoidably) suffers from the fact that Prolog is an untyped language. In Kluźniak [Klu93] arrays are present, as well, where they are called indexable sequences.

More recently, Barklund and Hill [BH95] proposed to add arrays and restricted quantification, a generalization of the bounded quantification, to Gödel, the programming language which does use types.

In the programs below we use bounded quantification, arrays and type declarations. The use of bounded quantifiers and arrays makes them simpler, more readable and closer to specifications. We declare constants, types, variables and relations in a style borrowed from the programming language Pascal. The choice of notation is preliminary.

We begin with two introductory examples.

Problem 3 Check whether a given sequence of 100 integers is ordered.

Solution

const `n = 100.`
rel `ordered:` **array** `[1..n]` **of integer**.
`ordered(A)` $\leftarrow$ $\forall$`I` $\in$ `[1..n-1] A[I]` $\leq$ `A[I+1].`

Comments This example shows that within the array subscripts terms should be evaluated, so that we can identify `1+1` with `2` etc. More precisely, "`+`" should be viewed here as an external procedure in the sense of Małuszyński et al. [MBB+93].

Note that the bounded universal quantifier $\forall$`I` $\in$ `[1..n]` does *not* correspond to the imperative **for** `i:=1` **to** n loop. The former is executed as long as a failure does not arise, i.e. up to n times, whereas the latter is executed precisely n times. The programming construct $\forall$`I` $\in$ `[1..n] Q` actually corresponds to the construct

for i:=1 **to** n **do if** $\neg$ Q **then**
  begin
    failure := **true**; *exit*
  end

which is clumsy and unnatural within the imperative programming paradigm.

(Feliks Kluźniak suggested to us the following, slightly more natural interpretation of $\forall$`I` $\in$ `[1..n] Q`:

i:=1;
while i $\leq$ n **cand** Q **do** i:=i+1;
failure := i $\leq$ n,

where **cand** is the "conditional **and**" connective (see Gries [Gri81, pages 68-70].))

Problem 4 Generate all members of a given sequence of 100 elements.

Solution

```
const n = 100.
rel member:  (*, array [1..n] of *).
member(X, Y) ← ∃I ∈ [1..n] X = Y[I].
```

Comments Here, Y is the given sequence. "*" stands for an unknown type. "=" is a built-in declared as

```
rel =:  (*, *).
DELAY X = Y UNTIL known(X) ∨ known(Y).
```

In other words, "=" is defined on any type and the calls to "=" are delayed until the value of one of its arguments is known, i.e. uniquely determined. If the values of both arguments are known, then it behaves like the usual comparison relation of Prolog and if the value of only one argument is known and the other is a, possibly subscripted, variable, then "=" behaves like the is built-in of Prolog. The case when one of the arguments is known and the other is not a variable does not arise here. known(X) is a built-in which holds when its argument is uniquely determined. It corresponds to ground(X) in Prolog.

This example shows the usefulness of polymorphic types in the presence of arrays. The bounded existential quantifier ∃I ∈ [1..n] implements backtracking and has no counterpart within the imperative programming paradigm.

Problem 5 Arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1,9]$ there are exactly i numbers between successive occurrences of i (see Coelho and Cotta [CC88, page 193]).

Solution

```
rel sequence:  array [1..27] of [1..9].
sequence(A) ← ∀I ∈ [1..9] ∃J ∈ [1..25-2I]
    (A[J] = I, A[J+I+1] = I, A[J+2I+2] = I)).
```

Comments The range J ∈ [1..25-2I] comes from the requirement that the indices J, J+I+1, J+2I+2 should lie within [1..27]. Thus J+2I+2 ≤ 27, that is J ≤ 25-2I.

Problem 6 Generate all permutations of a given sequence of 100 elements.

First we provide a solution for the case when there are no repeated elements in the sequence.

Solution 1

```
const n = 100.
rel permutation:  (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ← ∀I ∈ [1..n] ∃J ∈ [1..n] Y[J] = X[I].
```

Here, X is the given sequence. Alternatively,

```
permutation(X, Y) ← ∀I ∈ [1..n] member(X[I], Y).
```

Comments Note the similarity in the structure between this program and the one that solves problem 1. This program is incorrect when the sequence contains repeated elements. For example for n = 3 and X:= 0,0,1, Y:= 0,1,1 is a possible answer.

To deal with the general case we use local array declarations and reuse the above program.

Solution 2

```
const n = 100.
rel permutation: (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ←
    var A: array [1..n] of [1..n].
    ∀I ∈ [1..n] ∃J ∈ [1..n] A[J] = I,
    ∀I ∈ [1..n] Y[I] = X[A[I]].
```

Comments This solution states that A is an onto function from [1..n] to [1..n] and that a permution of a sequence of n elements is obtained by applying the function A to its indices.

Next, consider two well-known chess puzzles.

Problem 7 Place 8 queens on the chess board so that they do not check each other.

First, we provide a naive generate and test solution. It will be of use in the next section.

Solution 1

```
const n = 8.
type board: array [1..n] of [1..n].
rel queens, generate, safe: board.

queens(X) ← generate(X), safe(X).

generate(X) ← ∀I ∈ [1..n] ∃J ∈ [1..n] X[I] = J.

safe(X) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n]
    (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J)).
```

Comments To improve readability board is explicitly declared here as a type. Declaratively, this program states the conditions which should be satisfied by the values chosen for the queens. "≠" is a built-in declared as

```
rel ≠: (*, *).
```

In this section we use it only to compare terms with known values. Then "≠" behaves like the usual arithmetic inequality relation of Prolog. A more general usage of "≠" will be explained in the next section.

Next, we give a solution which involves backtracking.

Solution 2

```
const n = 8.
type board: array [1..n] of [1..n].
rel queens: board.

queens(X) ← ∀J ∈ [1..n] ∃K ∈ [1..n]
    (X[J] = K,
     ∀I ∈ [1..J-1]
        (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J))).
```

Comments Declaratively, this program states the conditions each possible value K for a queen placed in column J should satisfy.

Problem 8 Knight's tour. Find a cyclic route of a knight on the chess board so that each field is visited exactly once.

Solution We assign to each field a value between 1 and 64 and formalize the statement "from every field there is a "knight-reachable" field with the value one bigger". By symmetry we can assume that the value assigned to the field X[1, 1] is 1. Taking into account that the route is to be cyclic we actually get the following solution.

```
const n = 8.
type board: array [1..n, 1..n] of [1..n^2].
rel knight: board.
    go_on: (board, [1..n], [1..n]).

knight(X) ← ∀I ∈ [1..n] ∀J ∈ [1..n] go_on(X, I, J), X[1, 1] = 1.

go_on(X, I, J) ← ∃I1 ∈ [1..n] ∃J1 ∈ [1..n]
    (abs((I-I1)·(J-J1)) = 2, X[I1, J1] = (X[I, J] mod n^2) + 1).

DELAY go_on(X, I, J) UNTIL known(X[I,J]).
```

Comments Note that the equation abs(X · Y) = 2 used in the definition of go_on has exactly 8 solutions, which determine the possible directions for a knight move. Observe that each time this call to "=" is selected, both arguments of it are known. The efficiency of go_on could of course be improved by explicitly enumerating the choices for the offsets of the new coordinates w.r.t. the old ones.

The behaviour of the above program is quite subtle. First, thanks to the delay declaration, 64 constraints of the form go_on(X, I, J) are generated. Then, thanks to the statement X[1, 1] = 1, the first of them is "triggered" which one by one activates the remaining constraints. The backtracking is carried out by choosing different values for the variables I1 and J1. The delay declaration is not needed, but without it this program would be hopelessly inefficient.

It is interesting to note that in Wirth [Wir76], a classical book on programming in Pascal, the solutions to the last two problems are given as prototypical examples of recursive programs. Here recursion is implicit.

We conclude this section by one more program. It will be needed in the next section.

Problem 9 Let m = 50 and n = 100. Determine the number of different elements in an array X:**array** [1..m, 1..n] **of integer**.

Solution

```
const m = 50.
      n = 100.
type board:  array [1..m,1..n] of integer.
rel count:  (board, natural).

count(X, Number) ←
    Number = m · n -
     #(I, J: I ∈ [1..m], J ∈ [1..n]:
         (∃K ∈ [1..I-1] ∃L ∈ [1..n] X[I,J] = X[K,L])
            % X[I,J] occurs in an earlier row
          ∨ (∃L ∈ [1..J-1] X[I,J] = X[I,L]).
            % X[I,J] occurs earlier in the same row
       ).
```

Comments In this program we used the counting quantifier introduced in Gries [Gri81, page 74] and adopted in Kluźniak [Klu93] in the specification language SPILL-2. In general, given lists `L1, L2`, the term `#(I, J: I ∈ L1, J ∈ L2: Q)` stands for the number of pairs `(i,j)` such that `i ∈ L1, j ∈ L2` and for which the query `Q{I/i,J/j}` succeeds. It is possible to avoid the use of the counting quantifier at the expense of introducing a local array of type `board`. This alternative program is more laborious to write.

This concludes our presentation of selected logic programs written using arrays and bounded quantifiers. Other examples, including those involving numerical computation can be found in Barklund and Millroth [BM94].

4 Arrays and Bounded Quantifiers in Constraint Logic Programming

We now present some constraint logic programs. These are constraint programs with finite domains in the style of van Hentenryck [vH89]). Each of them has a similar pattern: constraints are first generated, and then resolved after the possible values for variables are successively generated. To clarify their use we provide here alternative solutions to two problems discussed in the previous section.

Problem 10 Solve problem 7 by means of constraints.

Solution

```
const n = 8.
type board:  array [1..n] of [1..n].
rel queens, safe, generate:  board.

queens(X) ← safe(X), generate(X).

safe(X) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n]
    (X[I] ≠ X[J], X[I] ≠ X[J] + (J-I), X[I] ≠ X[J] + (I-J)).
generate(X) ← ∀I ∈ [1..n] ∃J ∈ dom(X[I]) X[I] = J.
```

Comments Here dom(X), for a (possibly subscripted) variable X, is a built-in which denotes the list of current values in the domain of X, say in the ascending order. The value of dom(X) can change only by decreasing, by executing a constraint, so in the above program an atom of the form X $\neq$ t.

The relation "$\neq$" was used in the previous section only in the case when both arguments of it were known. Here we generalizes its usage, as we now allow that one or both sides of it are not known. In fact, "$\neq$" is a built-in defined as in van Hentenryck [vH89, pages 83-84], though generalized to arbitrary non-compound types.

We require that one of the following holds:

- Both sides of "$\neq$" are known. This case is explained in the previous section.
- At most one of the sides of "$\neq$" is known and one of the sides of "$\neq$", denoted below by X, is either a simple variable or a subscripted variable with a known subscript.

In the second case X $\neq$ t is defined as follows, where for a term s, Val(s) stands for the set of its currently possible values:

```
if Val(X) ∩ Val(t) = ∅ then succeed
elseif Val(t) is a singleton then % t is known, so X is not known
    begin dom(X):= dom(X) - Val(t); % dom(X) ≠ ∅
      if dom(X) = {f} then X:= f
    end .
```

If neither Val(X) $\cap$ Val(t) = $\emptyset$ nor Val(t) is a singleton, then the execution of X $\neq$ t is delayed. We treat t $\neq$ X as X $\neq$ t.

So for example in the program fragment

```
...
type bool:  [false, true].
var B: bool.
    A: array [1..2] of bool.
A[1] ≠ A[2], A[1] ≠ B, B = true.
...
```

the constraints A[1] $\neq$ A[2] and A[1] $\neq$ B are first delayed and upon the execution of the atom B = true the variable A[1] becomes false and A[2] becomes true.

In turn, in the case of the program given above the execution of an atom of the form X[I] = J for some I,J $\in$ [1..n] can affect the domains of the variables X[K] for K $\in$ [I+1..n]

This solution to the 8 queens problem is a forward checking program (see van Hentenryck [vH89, pages 122-127]). Note the textual similarity between this program and the one given in solution 1 to problem 7. Essentially, the calls to the safe and generate relations are now reversed. The generate relation corresponds to the labeling procedure in van Hentenryck [vH89]). In the subsequent programs the definition of the generate relation is always of the same format and is omitted.

Problem 11 Solve problem 6 by means of constraints.

Solution

```
const n = 100.
rel permutation: (array [1..n] of *, array [1..n] of *).
permutation(X, Y) ←
    type board: array [1..n] of [1..n].
    rel one_one, generate: board.

    one_one(Z) ← ∀I ∈ [1..n] ∀J ∈ [I+1..n] Z[I] ≠ Z[J].

    var A: board.
    one_one(A), generate(A),
    ∀I ∈ [1..n] Y[I] = X[A[I]].
```

Comments In this solution, apart from the local array declaration, we also use local type and relation declarations. The efficiency w.r.t. to the logic programming solution is increased by stating, by means of the call to the one_one relation, that A is a 1-1 function. This replaces the previously used statement that A is an onto function. The call to one_one generates $n \cdot (n-1)/2 = 4950$ constraints.

We conclude this section by dealing with another classic problem — that of colouring a map.

Problem 12 Given is a binary relation neighbour between countries. Colour a map in such a way that no two neighbours have the same color.

Solution

```
type color: [blue, green, red, yellow].
     countries: [austria, belgium, france, italy, ...].
rel map_color, constrain, generate: array countries of color.
    neighbour: (country, country).

map_color(X) ← constrain(X), generate(X).

constrain(X) ← ∀I ∈ countries ∀J ∈ countries
    neighbour(I,J) → X[I] ≠ X[J].
```

Comments We interpret here P → Q as follows:

```
(P → Q) ← P, Q.
(P → Q) ← ¬P.
```

so like the IF P THEN Q statement of Gödel. Note that in the above program at the moment of selection of the P → Q statement P is ground. Obviously, an efficient implementation of P → Q should avoid the reevaluation of P.

Thus the constrain relation generates here the constraints of the form X[I] ≠ X[J] for all I,J such that neighbour(I,J).

5 Adding Minimization and Maximization

Next, we introduce a construct allowing us to express in a compact way the requirement that we are looking for an optimal solution. To this end we introduce the *minimization operator* Y = μX:Q which is defined as follows:

```
Y = μX:Q  ←  Q{X/Y}, ¬(∃X ( X < Y, Q)).
```

We assume here that X and Y are of the same type and that $<$ is a built-in ordering on the domain of the type of X and Y. The existential quantifier ∃X Q is defined by the clause

```
∃X Q  ←  Q.
```

The efficient implementation of the minimization operator should make use of memoization (sometimes called tabulation) to store the solutions to the query Q found during the successive attempts to find a minimal one.

A dual operator, the *maximization operator* Y = νX:Q, is defined by:

```
Y = νX:Q  ←  Q{X/Y}, ¬ (∃X ( X > Y, Q)).
```

As before we assume that $>$ is a built-in ordering on the domain of the type of X and Y. In Barklund and Hill [BH95] the minimization and the maximization operators are introduced as a form of arithmetic quantifiers, in the style of the counting quantifier introduced earlier. The above two clauses show that they are derived concepts.

The following simple example illustrates the use of these constructs.

Problem 13 Find a minimum and a maximum of a given sequence of 100 integers.

Solution

```
const n = 100.
rel min_and_max:  (integer, integer, array [1..n] of integer).
min_and_max(Min, Max, A) ←
    Min = μX: ∃I ∈ [1..n] X = A[I],
    Max = νX: ∃I ∈ [1..n] X = A[I].
```

Next, we use these two operators in two constraint programs.

Problem 14 The cutting stock problem (see van Hentenryck [vH89, pages 181-187]). There are 72 configurations, 6 kinds of shelves and 4 identical boards to be cut. Given are 3 arrays:

```
Shelves:array [1..72, 1..6] of natural,
Req:array [1..6] of natural,
Waste:array [1..72] of natural.
```

Shelves[K,J] denotes the number of shelves of kind J cut in configuration K, Waste[I] denotes the waste per board in configuration I and Req[J] the required number of shelves of kind J. The problem is to cut the required number of shelves of each kind in such a way that the total waste is minimized.

Solution We represent the chosen configurations by the array

Conf: **array** [1..4] **of** [1..72]

where Conf[I] denotes the configuration used to cut the board I.

```
rel solve:  (array [1..4] of [1..72], natural).
    generate:  array [1..4] of [1..72].

solve(Conf, Sol) ←
    Sol = μTCost:
```

```
    % Sol is the minimal TCost such that:
∀I ∈ [1..3] Conf[I] ≤ Conf[I+1],
    % symmetry between the boards
∀J ∈ [1..6] Σ⁴_{I=1} Shelves[Conf[I],J] ≥ Req[J],
    % enough shelves are cut
TCost = Σ⁴_{I=1} Waste[Conf[I]],
    % TCost is the total waste
generate(Conf).
```

Comments In this program we used as a shorthand the sum notation "Σ ...". In general, it is advisable to use the sum quantifier (see Gries [Gri81, page 72]), which allows us to use $\Sigma^{l}_{I=k}$ t as a term. The sum quantifier is adopted in SPILL-2 language of Kluźniak [Klu93]. Kluźniak's notation for this expression is: `(S I: k ≤ I ≤ l: t)`. The interpretation of the constraints of the form `X ≤ t`, `X ≥ t` or `X = t` is similar to that of `X ≠ t` and is omitted.

We conclude by solving the following problem.

Problem 15 Let `m = 50` and `n = 100`. Given is an array `Co` which assigns to each pixel on an `m` by `n` board a colour. A *region* is a maximal set of adjacent pixels that have the same colour. Determine the number of regions.

In the program below we assign to the pixels belonging to the same region the same natural number, drawn between 1 and m·n. If we maximize the number of so used natural numbers we obtain the desired solution.

Solution

```
const m = 50.
      n = 100.
type color: [blue, green, red, yellow].
     pattern: array [1..m,1..n] of color.
     board: array [1..m,1..n] of [1..m·n].
rel pixel: (pattern, natural).
    no: (pattern, board).
    generate: board.
    count: (board, natural).

pixel(Co, Sol) ← Sol = νNumber:
    var X: board.
    no(Co, X), generate(X), count(X, Number).

no(Co, X) ← ∀I ∈ [1..m] ∀J ∈ [1..n]
     (
      (I < m → (Co[I,J] = Co[I+1,J] ↔ X[I,J] = X[I+1,J])),
      (J < n → (Co[I,J] = Co[I,J+1] ↔ X[I,J] = X[I,J+1]))
     ).
```

Comments The count relation is defined in the solution to problem 9. In the above program first $2m \cdot n - (m + n) = 9850$ constraints are generated. Each of them deals with two adjacent fields and has the form of an equality or inequality. Then the possible values for the elements of

X are generated and the number Number of so used natural numbers is counted. The maximum value for Number is then the desired solution.

The resulting program is probably not efficient, but still it is interesting to note that the problem at hand can be solved in a simple way without explicit recursion.

6 Conclusions

We have presented here several logic and constraint logic programs that use bounded quantification and arrays. We hope that these examples convinced the readers about the usefulness of these constructs. We think that this approach to programming is especially attractive when dealing with various optimization problems, as their specifications often involve arrays, bounded quantification, summation, and minimization and maximization. Constraint programming solutions to these problems can be easily written using arrays, bounded quantifiers, the sum and cardinality quantifiers, and the minimization and maximization operators. As examples let us mention the stable marriage problem, various timetabling problems and integer programming.

Of course, it is not obvious whether the solutions so obtained are efficient. We expect, however, that after an addition of a small number of built-in's, like deleteff and deleteffc of van Hentenryck [vH89, pages 89-90], it will be possible to write simple constraint programs which will be comparable in efficiency with those written in other languages for constraint logic programming.

When introducing arrays we were quite conservative and only allowed static arrays, i.e. arrays whose bounds are determined at compile time. Of course, in a more realistic language proposal also open arrays, i.e. arrays whose bounds are determined at run-time should be allowed. One might also envisage the use of flexible arrays, i.e. arrays whose bounds can change at run-time.

In order to make this programming proposal more realistic one should provide a smooth integration of arrays with recursive types, like lists and trees. In the language SPILL-2 of Kluźniak [Klu93] types are present but only as sets of ground terms, and polymorphism is not allowed. Barklund and Hill [BH95] proposed to add arrays to Gödel (which does support polymorphism) as a system module. We would prefer to treat arrays on equal footing with other types.

We noticed already that within the logic programming paradigm the demarkation line between iteration and recursion differs from the one in the imperative programming paradigm. In order to better understand the proposed programming style one should first clarify when to use iteration instead of recursion. In this respect it is useful to quote the opening sentence of Barklund and Millroth [BM94]: "Programs operating on inductively defined data structures, such as lists, are naturally defined by recursive programs, while programs operating on "indexable" data structures, such as arrays, are naturally defined by iterative programs".

We do not entirely agree with this remark. For example, the "suffix" quantifiers mentioned in Section 2 allow us to write many list processing programs without explicit use of recursion (see Voronkov [Vor92]) and the quicksort program written in the logic programming style is more natural when written using recursion than iteration.

The single assignment property of logic programming makes certain programs that involve arrays (like Warshall's algorithm) obviously less space efficient than their imperative programming counterparts. This naturally motivates research on efficient implementation techniques of arrays within the logic programming paradigm.

Finally, a comment about the presentation. We were quite informal when explaining the meaning of the proposed language constructs. Note that the usual definition of SLD-resolution has to be appropriately modified in presence of arrays and bounded quantification. For example, the query `X[1] = 0, ∀I ∈ [1..2] X[I] ≠ 0` fails but this fact can be deduced only when the formation of resolvents is formally explained. To this end substitution for subscripted variables needs to be properly defined. One possibility is to adopt one of the definitions used in the context of verification of imperative programs (see Apt [Apt81, pages 460-462]). We leave the task of defining a formal semantics to another paper.

Acknowledgements I would like to thank here Jonas Barklund and Feliks Kluźniak for useful discussions on the subject of bounded quantification and Pascal van Hentenryck for encouragement at the initial stage of this work. Also, I am grateful to Feliks Kluźniak for helpful comments on this paper.

References

[Apt81] K.R. Apt. Ten years of Hoare's logic, a survey, part I. *ACM TOPLAS*, 3:431–483, 1981.

[BB93] J. Barklund and J. Bevemyr. Prolog with arrays and bounded quantifications. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning—Proc. 4th Intl. Conf.*, LNCS 698, pages 28–39, Berlin, 1993. Springer-Verlag.

[BH95] J. Barklund and P. Hill. Extending Gödel for expressing restricted quantifications and arrays. UPMAIL Tech. Rep. 102, Computer Science Department, Uppsala University, Uppsala, 1995.

[BM94] J. Barklund and H. Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. UPMAIL Tech. Rep. 71, Computer Science Department, Uppsala University, Uppsala, 1994.

[CC88] H. Coelho and J. C. Cotta. *Prolog by Example.* Springer-Verlag, Berlin, 1988.

[ER84] L.-H. Eriksson and M. Rayner. Incorporating mutable arrays into logic programming. In S. Å. Tarnlund, editor, *Proceedings of the 1991 International Conference on Logic Programming*, pages 101–114. Uppsala University, 1984.

[Gri81] D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

[Har80] D. Harel. And/or programs: a new approach to structured programming. *ACM Toplas*, 2(1):1–17, 1980.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* The MIT Press, 1994.

[Klu91] F. Kluźniak. Towards practical executable specifications in logic. Research report LiTH-IDA-R-91-26, Department of Computer Science, Linköping University, August 1991.

[Klu93] F. Kluźniak. SPILL-2: the language. Technical report ZMI Reports No 93-03, Institute of Informatics, Warsaw University, July 1993. A deliverable for year 1 of the BRA Esprit Project Compulog 2.

[KM94] F. Kluźniak and M. Miłkowska. Readable, runnable requirements specifications: Bridging the credibility gap. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming. Proceedings of the 6th International Symposium, PLILP'94. Madrid, September 1994*, pages 449–450. Springer-Verlag, 1994.

[MBB+93] J. Małuszyński, S. Bonnier, J. Boye, F. Kluźniak, A. Kågedal, and U. Nilsson. Logic programs with external procedures. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Current Trends in Logic Programming Languages*, pages 21–48. The MIT Press, Cambridge, Massachussets, 1993.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[Vor92] A. Voronkov. Logic programming with bounded quantifiers. In A. Voronkov, editor, *Logic Programming and Automated Reasoning—Proc. 2nd Russian Conference on Logic Programming*, LNCS 592, pages 486–514, Berlin, 1992. Springer-Verlag.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.