Joint NCC & IIS Bull., Comp. Science, 16 (2001), 103–113

Constraint propagation in presence of arrays

S. Brand

We describe the use of array expressions as constraints, which represents a consequent generalisation of the `element` constraint. Constraint propagation for array constraints is studied theoretically, and for a set of domain reduction rules the local consistency they enforce, arc-consistency, is proved. An efficient algorithm is described that encapsulates the rule set and so inherits the capability to enforce arc-consistency from the rules.

1. Introduction

Many problems can be modelled advantageously using "look up" functionality: to access an object given its index. Imperative programming languages provide arrays for this. With `i` one of `1`, `2`, `3` and a definition such as `integer a[3]`, the construct `a[i]` represents an integer variable, while with a definition `b[] = {5, 7, 9}` the 'value' of `i` according to table `b` can be looked up by `x = b[i]`.

A usual condition for look-up expressions to be valid is that the index be known when the expression is evaluated. In a constraint programming environment, this is a restriction that can be disposed of. The binary `element` constraint (originally in CHIP, [4]), semantically equivalent to a lookup expression using a 1-dimensional array, allows a variable as the index and a variable for the result, constraining both. It proved to be very beneficial to allow a variable for the index. Many important problems (scheduling, resource allocation, etc.) modelled as CSPs make use of this constraint.

OPL, a modelling language for combinatorial optimisation and constraint programming ([11]), supports arrays of constants and variables, and arrays indexed by variables (or other expressions). These array expressions are most general. However, domain reduction in OPL is weaker than possible, for instance the reduction for an index variable depends on its position ([11], p. 100).

In this work we study constraint propagation enforcing arc-consistency for general array expressions. Arrays are multidimensional, and variables can occur wherever constants can. An expression $x = a[y_1, \dots, y_n]$ is seen as a constraint on the variables x and $y_1, \dots, y_n$, and all the variables collected in the array a.

Example. Consider an application of arrays. Assume a conventional crossword grid, with entries for words in the rows and columns and remaining fields blackened. Further consider a set of words, a subset of which is to be filled into the entries in the grid. A natural formulation of this problem as a CSP is to take, for each word entry, a variable E_i whose initial domain is the set of words that fit the entry length-wise.

The words, split up into their letters, are collected in a two-dimensional array l such that $l[w, p]$ represents the letter of word w at position p. The conditions on crossings of entries are then easily stated as constraints. A crossing of field E_1 at position 4 and field E_2 at position 3 is stated as $l[E_1, 4] = l[E_2, 3]$. An additional `alldifferent` constraint on the E_i ensures that no two word entries contain the same word.

Enforcing arc-consistency for array expressions solves some instances of the crossword problem without any backtracking ([10], p. 140, which uses a custom constraint for crossing entries that is equivalent to the one here).

2. Preliminaries

A constraint satisfaction problem $\langle \mathcal{C}; \mathcal{D} \rangle$ consists of a set of variables (implicit here), a set $\mathcal{D}$ of domain expressions $x \in D_x$ that associate a set of admissible values with every variable, and a set $\mathcal{C}$ of constraints. A constraint is a relation on a set of variables that is a subset of the cartesian product of their domains.

A *solution* for a constraint is an assignment of values to its variables that are consistent with the constraint. A solution for a CSP is an assignment that is a solution for all its constraints. A CSP, or a constraint, is *satisfiable* if a solution exists. A domain value, or a partial solution, is *supported* if it is part of a solution.

Local consistency notions, weaker approximations of (global) satisfiability, are essential in constraint solving. A central one is arc-consistency ([8]). We disregard the arity of constraints and define: a constraint is *arc-consistent*, if all domain values of all its variables are supported. A CSP is arc-consistent if all its constraints are arc-consistent.

2.1. A rule-based formalism

Constraint programming can be seen as transforming a CSP into one or several simpler but equivalent CSPs in a rule-based way. This view allows separate consideration of the reductive strength of some set of constraint propagation rules and its scheduling. The transformations on CSPs lend themselves to a declarative formulation. We adopt the proof theoretic formalism of [1], and introduce the relevant elements.

A transformation step from a CSP $\mathcal{P}$, the premise, to a CSP $\mathcal{Q}$, the conclusion, by application of a rule (r) and possibly subject to a side condition $\langle C \rangle$ on $\mathcal{P}$ is represented as

$$\text{(r)} \quad \frac{\mathcal{P}}{\mathcal{Q}} \quad \langle C \rangle$$

Two CSPs $\mathcal{P}$ and $\mathcal{Q}$ are *equivalent* if all variables in $\mathcal{P}$ are present in $\mathcal{Q}$ and every solution for $\mathcal{P}$ can be extended to a solution for $\mathcal{Q}$ by an assignment to variables only in $\mathcal{Q}$. If a rule application preserves equivalence, then the rule is *sound*.

A rule is required to be relevantly *applicable*, that is, the result $\mathcal{Q}$ must be different from $\mathcal{P}$ in the sense that the set of domain expressions or the set of constraints changes. If a rule, or a set of rules, is not applicable to $\mathcal{P}$ then $\mathcal{P}$ is *stable* or *closed under* it. Applying a rule to a constraint means applying it to the CSP consisting only of this constraint.

Notation. The domain expressions $v \in D_v$ used in the rules are implicitly represented in the set $\mathcal{D}$. Replacing a domain expression from $\mathcal{D}$ is denoted by $\mathcal{D}, v \in D_v^{new}$. If in $\mathcal{P} = \langle \mathcal{C}; \mathcal{D} \rangle$ the set of constraints consists of only one constraint, $\mathcal{C} = \{con\}$, then we may just write $\mathcal{P} = \langle con; \mathcal{D} \rangle$. The expression $s \mapsto t$ denotes a substitution, assignment, or mapping, from s to t.

Example. We illustrate these concepts with a rule-based characterisation of arc-consistency. A constraint C is arc-consistent if for all variables v of C and all values $d \in D_v$ an instantiation of v to d in C, written $\mathcal{C}\{v \mapsto d\}$, retains satisfiability. If $\mathcal{C}\{v \mapsto d\}$ is not satisfiable, then d is redundant and can be removed from D_v. The resulting CSP is equivalent to the original one. This principle is captured in a rule:

Lemma 1. *A satisfiable constraint C is arc-consistent iff it is closed under the application of*

$$\text{(ac)} \qquad \frac{\langle C\,;\,\mathcal{D} \rangle}{\langle C\,;\,\mathcal{D}, v \in D_v \setminus \{d\} \rangle} \quad \mathcal{C}\{v \mapsto d\} \textit{ has no solution}$$

□

3. Arc-consistency for array constraints

An array a of arity n is a set of mappings *index* $\mapsto$ *variable*, where *index* is a unique n-tuple of constants, and *variable* is a variable with a domain.

The array expression $a[b_1, \ldots, b_n]$ evaluates to v, if a contains a mapping $(b_1, \ldots, b_n) \mapsto v$, otherwise it is not defined (in what follows it is assumed that indices accessing a are valid). Note that arrays of constants come as a specialization of this model.

3.1. Simple array constraints

Array expressions $a[y_1, \ldots, y_n]$ are functional. The simplest extension to a constraint is the equality constraint $C \equiv \langle x = a[y_1, \ldots, y_n]\rangle$. We establish arc-consistency first for this case, and then discuss compound (nested) array expressions. Also, occurrences of variables are restricted in that no variable in the constraint may occur more than once (C is linear). Note that the variables of C are x, $y_1, \ldots, y_n$, and all variables v for valid $(b_1, \ldots, b_n)$ and $(b_1, \ldots, b_n) \mapsto v$ in a. Such v will from now on be denoted directly as $a[b_1, \ldots, b_n]$.

Theorem 1 (Arc-consistency for arrays). *A satisfiable linear equality constraint $\langle x = a[y_1, \ldots, y_n]\rangle$ is arc-consistent iff it is closed under the rule set $\mathcal{R}_{\mathrm{arr}}$:*

$$(\mathrm{arr}_x) \qquad \frac{\langle x = a[y_1, \ldots, y_n] \,;\, \mathcal{D}\rangle}{\left\langle x = a[y_1, \ldots, y_n] \,;\, \mathcal{D}, x \in D_x \cap \left(\bigcup_{b_i \in D_{y_i}} D_{a[b_1,\ldots,b_n]}\right)\right\rangle}$$

$$(\mathrm{arr}_y) \qquad \frac{\langle x = a[y_1, \ldots, y_n] \,;\, \mathcal{D}\rangle}{\langle x = a[y_1, \ldots, y_n] \,;\, \mathcal{D}, y_k \in D_{y_k} \backslash \{b\}\rangle} \quad \langle C_y\rangle$$

$$\langle C_y\rangle : \qquad D_x \cap \left(\bigcup_{b_i \,\in\, D_{y_i},\; b_k = b} D_{a[b_1,\ldots,b_n]} \right) = \varnothing$$

$$(\mathrm{arr}_a) \qquad \frac{\langle x = a[y_1, \ldots, y_n] \,;\, \mathcal{D}\rangle}{\left\langle x = a[y_1, \ldots, y_n] \,;\, \mathcal{D}, a[b_1, \ldots, b_n] \in D_{a[b_1,\ldots,b_n]} \cap D_x\right\rangle} \quad \langle C_a\rangle$$

$$\langle C_a\rangle : \qquad D_{y_1} \times \ldots \times D_{y_n} = \{(b_1, \ldots, b_n)\}$$

□

Proof.

($\Leftarrow$) Suppose $C \equiv \langle x = a[y_1, \ldots, y_n]\rangle$ is closed under $\mathcal{R}_{\mathrm{arr}}$. Then all values in the domains of variables in C are supported.
(1) Take some $d \in D_x$. C is closed under (arr_x), thus also $d \in \left(\bigcup_{b_i \in D_{y_i}} D_{a[b_1,\ldots,b_n]}\right)$. Then there exists some $(b_1, \ldots, b_n)$ with $d \in D_{a[b_1,\ldots,b_n]}$. This index and $a[b_1, \ldots, b_n] \mapsto d$ support $x \mapsto d$.

(2) For some $b \in D_{y_k}$ we consider the necessarily failing condition of (arr_y). Thus a value d exists in both D_x and $D_{a[b_1,\dots,b_n]}$, for some $(b_1,\dots,b_n)$ with $b_k = b$. Assigning b_i to y_i, and $x \mapsto d$ and $a[b_1,\dots,b_n] \mapsto d$ is a solution supporting b.
(3) Consider a value $d \in D_{a[b_1,\dots,b_n]}$ and the following cases:
(3.1) $(b_1,\dots,b_n) \notin D_{y_1} \times \dots \times D_{y_n}$.
The index $(y_1,\dots,y_n)$ can not select the variable $a[b_1,\dots,b_n]$; however, C remains satisfiable. Therefore, there is a solution for C that is indifferent to the value of $a[b_1,\dots,b_n]$, and so supports $a[b_1,\dots,b_n] \mapsto d$.
(3.2) $(b_1,\dots,b_n) \in D_{y_1} \times \dots \times D_{y_n}$.
(3.2.1) $\{(b_1,\dots,b_n)\} = D_{y_1} \times \dots \times D_{y_n}$.
Here the condition of (arr_a) is fulfilled, its consequence holds and according to it $d \in D_x$. A supporting solution is therefore $x \mapsto d$, $a[b_1,\dots,b_n] \mapsto d$, $y_i \mapsto b_i$ for all i.
(3.2.2) Some D_{y_k} contains more than one element.
Consider some index $(b'_1,\dots,b'_n)$ with $b'_k \neq b_k$ that also fulfills $D_x \cap D_{a[b'_1,\dots,b'_n]} \neq \varnothing$. Such an index exists because otherwise (arr_y) would be applicable. Choose $d' \in D_x$ and instantiate $x \mapsto d'$, $a[b'_1,\dots,b'_n] \mapsto d'$, $y_i \mapsto b'_i$ for all i. This solution to C does not assign to $a[b_1,\dots,b_n]$ and hence supports $a[b_1,\dots,b_n] \mapsto d$.

($\Rightarrow$) Suppose here that C is not closed under $\mathcal{R}_{\text{arr}}$. Then domains of some variables in C contain unsupported values.
(1) Assume (arr_x) is applicable, that is, $D_x \supset \bigcup_{b_i \in D_{y_i}} D_{a[b_1,\dots,b_n]}$. Then there is some value $d \in D_x, d \notin D_{a[b_1,\dots,b_n]}$ for all $(b_1,\dots,b_n) \in D_{y_1} \times \dots \times D_{y_n}$. Clearly, d is not a part of any solution.
(2) Suppose some $b_k \in D_{y_k}$ could be removed by (arr_y). From the condition of (arr_y) it follows that with $y_k \mapsto b_k$ no index $(b_1,\dots,b_n)$ can be found that allows the same value for x and $a[b_1,\dots,b_n]$.
(3) For a singleton index domain and a possible application of (arr_a) , we consider $a[b_1,\dots,b_n] \mapsto d$ with $d \notin D_x$. Such d can not be supported by x. □

Linearity requirement. It is necessary to restrict occurrences of variables. Consider the array $xor = \{(0,0) \mapsto 0, \{(0,1) \mapsto 1, \{(1,0) \mapsto 1, \{(1,1) \mapsto 0\}$ and the CSP $\mathcal{P} \equiv \langle 0 = xor[y,y]; \{y \in \{0,1\}\}\rangle$. $\mathcal{P}$ is inconsistent but stable under $\mathcal{R}_{\text{arr}}$.

Origin of $\mathcal{R}_{\text{arr}}$. Each rule in $\mathcal{R}_{\text{arr}}$ can be derived as an instance of the general rule (ac) in Lemma 1. Such a derivation, perhaps unsurprisingly, proceeds along the same case distinctions as in the ($\Leftarrow$) part of the above proof. We believe the derivation to be interesting in its own right, but choose here the proof for its relative brevity.

3.2. Arc-consistency and compound expressions

The following result allows decomposition of nested array expressions and equality constraints for the purpose of establishing arc-consistency. Expressions such as $l[w,p] = l[w',p']$ from the crossword example are decomposed with a fresh variable into $v = l[w,p]$ and $v = l[w',p']$, upon which arc-consistency can be enforced independently.

Lemma 2. *Assume $C_t \equiv \langle s = t(v)\rangle$ and $C_v \equiv \langle v = r\rangle$ be linear constraints on, apart from v, distinct sets of variables. The constraint $C \equiv \langle s = t\{v \mapsto r\}\rangle$ is arc-consistent if C_t and C_v are arc-consistent.* □

Proof. Suppose C_t and C_v are arc-consistent.

Any solution for C_t assigns a value to v that is also supported by a solution to C_v, and vice versa. Due to the conditions on variables, such solutions do not assign to the same variables. Therefore, their union is also a solution for C. Thus, a supporting solution for any domain value of a variable in C_t, C_v, and C, can be extended to a supporting solution for C.

Hence, C is arc-consistent. □

3.3. Domain reduction and transformation

As instances of (ac), the rules in $\mathcal{R}_{\mathrm{arr}}$ are domain reduction rules by type. From a semantical, and particularly from an operational, point of view, however, it may be worth to have, instead transformation, the rules that change the representation of constraints.

Consider (arr_a), which applies if the index is fully instantiated. This means that we can also dispense entirely with the array look-up: no choice is left. The array expression can be replaced by the selected variable. Thus, an alternative to (arr_a) would be

$$\frac{\langle x = a[y_1, \ldots, y_n]\,;\, \mathcal{D}\rangle}{\left\langle x = a[b_1, \ldots, b_n]\,;\, \mathcal{D}, a[b_1, \ldots, b_n] \in D_{a[b_1,\ldots,b_n]} \cap D_x \right\rangle} \quad \langle C_a\rangle.$$

This rule is now both a transformation rule and a domain reduction rule. Note that the domain reduction takes place between variables. In the presence of the rules for primitive equality constraints $\langle x = y\rangle$, one can simplify even more into a pure transformation rule:

$$(\mathrm{arr}'_a) \qquad \frac{\langle x = a[y_1, \ldots, y_n]\,;\, \mathcal{D}\rangle}{\langle x = a[b_1, \ldots, b_n]\,;\, \mathcal{D}\rangle} \quad \langle C_a\rangle.$$

The combination of (arr'_a) and rules for $\langle x = y\rangle$ is equivalent to (arr_a).

4. A non-naive algorithm

An exhaustive application of $\mathcal{R}_{\text{arr}}$ is computationally expensive, in part unavoidable due to the strength of arc-consistency, and the large number of variables involved in array constraints. Inefficiency that can be remedied is the large number of set operations on domains, due to the fact that individual array variable domains $D_{a[b_1,\ldots,b_n]}$ are read and processed many times.

The algorithm arr-ac (Figure 1) reads every $D_{a[b_1,\ldots,b_n]}$ addressable by $(y_1,\ldots,y_n)$ at most once. Consider $T = D_x \cap D_{a[b_1,\ldots,b_n]}$ for some $(b_1,\ldots,b_n) \in D_{y_1} \times \ldots \times D_{y_n}$. T is a subset of the intersection in the conclusion of (arr_x), so it is necessarily a part of the new domain of x, and only $D_x \backslash T$ instead of D_x needs to be subjected to further restriction. With regard to (arr_y), a nonempty T implies that the side condition fails. Thus, no b_k of $(b_1,\ldots,b_n)$ can be removed from D_{y_k} by (arr_y).

for all i: $B_i := D_{y_i}$	index
for all i: $Y_i := D_{y_i}$	potentially redundant for y_i
$X := D_x$	potentially redundant for x
$S := \varnothing$	indices skipped for X
while $B \neq \varnothing$ and some $Y_k \neq \varnothing$ ¿	loop for Y and X
choose and remove $(b_1,\ldots,b_n)$ from B	
if some $b_k \in Y_k$ **then**	
$T := D_x \cap D_{a[b_1,\ldots,b_n]}$	
if $T \neq \varnothing$ **then**	
for all i: $Y_i := Y_i \backslash \{b_i\}$	
$X := X \backslash T$	
else $S := S \cup \{(b_1,\ldots,b_n)\}$	
while $S \neq \varnothing$ and $X \neq \varnothing$	rest loop for X
choose and remove $(b_1,\ldots,b_n)$ from S	
$X := X \backslash D_{a[b_1,\ldots,b_n]}$	
for all i: $D_{y_i} := D_{y_i} - Y_i$	remove redundant values
$D_x := D_x - X$	

Figure 1. arr-ac (core)

Note that arr-ac makes a positive guess whether the values are supported. If in the end some domain is really reduced, then arr-ac needs to repeat the run. Indeed, if somewhere before the regular end of the run, as described in Figure 1, it is definite that some domain D_{y_j} will be reduced, the run could terminate immediately, commit the change to D_{y_j}, and restart.

The core part of arr-ac can itself be regarded as a complex domain reduction rule, encapsulating (arr_x) and (arr_y). The rule set $\{\textsf{arr-ac}\text{:core}, (\text{arr}_a)\}$ establishes arc-consistency.

Example. Consider $x \in \{\mathsf{B}, \mathsf{C}, \mathsf{D}\}, y_1 \in \{1, 2\}, y_2 \in \{1, 2, 3\}$ and $\langle x = a[y_1, y_2] \rangle$ and let a be defined as an array of constants

(y_1, y_2)	1	2	3
1	A	B	C
2	D	E	F

The constraint is arc-consistent, which arr-ac verifies as follows. First it reads $a[1, 1] = \mathsf{A}$. Nothing is done. It follows that $a[1, 2] = \mathsf{B}$. B is in D_x, so B is a supported value for x, and 1 is supported for y_1 and 2 for y_2. The next step is reading $a[1, 3] = \mathsf{C}$. This supports C for x and 3 for y_2. Finally, $a[2, 1] = \mathsf{D}$ is reached. This supports the last missing value D for x, and, moreover, 2 for y_1 and 1 for y_2.

The support for all values in the domains was found, hence arc-consistency is established. Only one incomplete run was necessary, skipping the indices $(2, 2), (2, 3)$ that are still permissible by (y_1, y_2). □

For one run of arr-ac (and ignoring X here), the number of iterations that enter the computation of T has an upper bound of $\mathcal{O}(d^n)$ with d the maximal size of the domains of y_i. This reflects the number of possible different indices $(b_1, \ldots, b_n)$. The lower bound, on the other hand, is only $\mathcal{O}(d)$. It is reached when every iteration reduces all (nonempty) Y_i by an element, and occurs if the constraint is arc-consistent and every instantiation of $(y_1, \ldots, y_n)$ is part of a solution.

An operationally useful side effect of arr-ac is that it can also yield the variables that contain the supporting values. Initially, all variables $a[b_1, \ldots, b_n]$ are part of the constraint, whereas after complete instantiation of the index $(y_1, \ldots, y_n)$ only the variable $a[y_1, \ldots, y_n]$ is constrained and contains support. The algorithm arr-ac regards these variables $a[b_1, \ldots, b_n]$ as supporting, for which the intersection T is nonempty.

The algorithm arr-ac has been implemented in ECL^iPS^e ([6]), using the finite domain primitives of `lib(fd)`. An implementation of $\mathcal{R}_{\text{arr}}$ in the same environment Has been compared to arr-ac by testing it against an instance of the crossword problem and was roughly 50% slower.

5. Final remarks

5.1. Related work

The established precursor of array constraints is the `element` constraint ([4]). It is the one-dimensional specialization, and usually the look-up list that links an index with a result is restricted to a list consisting of constants.

Arrays in OPL ([11, 7]) are similarly general as in this work. In [9] on OPL++, a model of the stable marriages problem is described that employs an array of variables indexed by a variable. Constraint propagation of array expressions in OPL is strictly weaker, however. For all three cases treated by $\mathcal{R}_{\text{arr}}$, we could construct simple examples using small 2-dimensional arrays in which reduction of domains is possible but not performed, see Figures 2 and 3.

The paper [3] describes an implementation of `element` using indexicals in AKL(FD), in which the look-up list can consist of domain variables. It is equivalent to a one-dimensional instance of $\mathcal{R}_{\text{arr}}$.

In [2] a new constraint `case` is proposed that subsumes multidimensional array constraints with arrays of constants. An algorithm, which seems similar in effect to the use of $\mathcal{R}_{\text{arr}}$, based on graph theory, is outlined.

The paper [5], on unifying optimization and constraint satisfaction methods, studies a continuous relaxation of `element` with a look-up list of variables with continuous domains by using a cutting-planes approach.

5.2. Conclusions

We study here the use of arrays in constraint programming mainly from the theoretical point of view. There are good arguments in favour of the fact that arrays are beneficial in constraint models. Indices of objects are a basic notion in mathematics. The object `element` is implemented in many constraint systems. Arrays with multiple dimensions have long been used in imperative, now object-oriented, languages. These language styles obviously inspired OPL ([11] and [9]), a successful constraint programming system. Yet it would be desirable to have better examples of using the multidimensional arrays.

Such problems could also help us to evaluate the use of arc-consistency as the objective in constraint propagation. It is now clear from our experience that the type of consistency, that is most advantageous, depends on the problem. Sometimes a weaker notion, such as bound or range consistency, might suffice, for example, when applied in the early stages of solving a problem and later replaced by full arc-consistency. $\mathcal{R}_{\text{arr}}$ provides a starting point for obtaining the reduction rules for those consistency notions which are subsumed by, yet very similar to, arc-consistency.

Acknowledgements. Krzysztof Apt has suggested the topic of this work and made many helpful comments. I am thankful also to referee comments on an earlier presentation of the subject.

References

[1] K. R. Apt. A proof theoretic view of constraint programming. *Fundamenta Informatica*, 34, 1998.

[2] N. Beldiceanu. Global constraints as graph properties on structured network of elementary constaints of the same type. Technical report, Swedish Institute of Computer Science, Jan. 2000.

[3] B. Carlson, S. Haridi, and S. Janson. AKL(FD)-A concurrent language for FD programming. In M. Bruynooghe, editor, *Proceedings of the 11th International Symposium on Logic Programming*, Cambridge, MA, USA, 1994. MIT Press.

[4] M. Dincbas, P. V. Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In I. for New Generation Computer Technology (ICOT), editor, *International Conference on Fifth Generation Computer Systems*, volume 2. Springer Verlag, 1988.

[5] J. N. Hooker, G. Ottosson, E. S. Thorsteinsson, and H.-J. Kim. A scheme for unifying optimization and constraint satisfaction methods. *Knowledge Engineering Review, Special Issue on Artifical Intelligence and Operations Research*, 15(1):11–30, 2000.

[6] IC-Parc. *ECLiPSe*. `http://www.icparc.ic.ac.uk/eclipse/`.

[7] ILOG. *OPL Studio 3*. `http://www.ilog.com/`.

[8] A. K. Macworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):118–126, 1977.

[9] L. Michel and P. V. Hentenryck. OPL++: A modeling layer for constraint programming libraries. Technical Report CS-00-07, Department of Computer Science, Brown University, Dec. 2000.

[10] P. van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[11] P. van Hentenryck, I. Lustig, L. Michel, and J.-F. Puget. *The OPL optimization programming language*. M.I.T. Press, 1999.

```
enum      Dz  { i, j    };
enum      Dy  { k, l, m };
enum      Da  { p, q, r };

Da        a[Dz, Dy] = #[  i:  #[k:p, l:q, m:r]#,
                          j:  #[k:p, l:q, m:r]#   ]#;
var       Da  x;
var       Dz  z;
var       Dy  y;
var       Dz  u;
var       Dy  v;

solve {   v <> l;         //              OPL       arc-consistency
          a[u, v] = x;    //  x  in   { p, q, r }      { p, r }
                          //
          a[z, y] = q;    //  y  in   { k, l, m }       { l }
};
```

Figure 2. OPL: non-applied (arr_x), (arr_y)

```
enum      Dy  { i, j, k };
enum      Da  { p, q, r };

var       Da  a[Dy];
var       Da  x;
var       Dy  y;

solve {   y = j;
          x <> q;       //                OPL       arc-consistency
          x = a[y];     //  a[j]  in  { p, q, r }      { p, r }
};
```

Figure 3. OPL: non-applied (arr_a)