

An Analysis of Permutations in Arrays

Valentin Perrelle, Nicolas Halbwachs

▶ **To cite this version:**

Valentin Perrelle, Nicolas Halbwachs. An Analysis of Permutations in Arrays. Verification, Model Checking, and Abstract Interpretation, Jan 2010, Madrid, Spain. pp.279-294, 10.1007/978-3-642-11319-2_21 . hal-00456558

HAL Id: hal-00456558
https://hal.science/hal-00456558

Submitted on 15 Feb 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An analysis of permutations in arrays *

Valentin Perrelle and Nicolas Halbwachs
{Valentin.Perrelle,Nicolas.Halbwachs}@imag.fr

Vérimag**, Grenoble University – France

Abstract. This paper is concerned with the synthesis of invariants in programs with arrays. More specifically, we consider properties concerning array contents *up to a permutation*. For instance, to prove a sorting procedure, one has to show that the result is sorted, but also that *it is a permutation of the initial array*. In order to analyze this kind of properties, we define an abstract interpretation working on multisets of values, and able to discover invariant equations about such multisets.

1 Introduction

The analysis of properties of data structures is a challenging goal. It has been widely studied, but still strongly needs to be improved, concerning the efficiency and precision of analyzes, but also the class of properties that can be handled. Roughly speaking, data structure properties can be divided into three classes: (1) The most widely studied properties are *structural properties*: they concern the shape of data structures and the correctness of their accesses, independently of their contents. Array bound checking (e.g., [CC76,LS79]), and shape analysis (e.g., [WSR00,SRW02,DRS03]) address this class of properties. (2) More recently, several methods were proposed for analyzing *positional properties* of data structure contents, i.e., properties relating the value of a cell with its position in the structure [BMS06,IHV08,FQ02,LB04,JM07,GMT08,Cou03,GRS05,HP08]. The fact that two arrays are pointwise equal, or that a list is sorted, are examples of such properties. (3) In this paper, we will consider an instance of *non positional* properties, which concern the whole content of a data structure, independently of the structure itself. A typical example is the fact that an array is a permutation of another array. Showing that the result of a sorting procedure is indeed sorted is not enough to prove the procedure; one has to show also that the result is a permutation of the initial structure. There are many examples of such algorithms which are intended to reorganize a data structure without changing its global content. Showing that the global content is not changed is therefore an issue. Such non positional properties are not easily expressible with usual formalisms: they cannot be expressed as "$\forall \ldots \exists \ldots$" formulas as those considered in [SG09], and [SJ80] remarks that the fact that two arrays are equal *up*

* This work has been partially supported by the ASOPT project of the "Agence Nationale de la Recherche" of the French Ministry of Research

** Verimag is a joint laboratory of Université Joseph Fourier, CNRS and Grenoble-INP.

to a permutation cannot be expressed by a first order formula. This may explain that these properties have been more or less ignored in the literature on program analysis.

In this paper, we describe an analysis technique able to discover equations about global contents of arrays. Such a global content is a multiset, since several cells may contain the same value. For simplicity, the paper is specialized on (one-dimensional) array analysis, but the approach could be extended to other data structures. For instance, our method is able to discover that if, at the beginning of an insertion sort procedure, the array A to be sorted contains a multiset $\mathcal{M}$ of values, it contains the same multiset at the end of the procedure. Combined with an analysis of positional properties such as [HP08], it provides an automatic method for *discovering* the exact input/output behavior of the procedure.

Basically, our analysis is an abstract interpretation propagating multiset equations. It is helped by other, more classical analyzes, discovering equalities and disequalities between indexes — which are, of course, very important to deal with aliases[1] — and equalities between variables and array cells. After giving the basic notations and definitions (Section 2), we introduce the analysis by an example (Section 3). Section 4 presents the principles of the analysis, before addressing the main problem, which concerns the computation of an upper bound of two abstract values.

Section 5 presents a first version of our abstract lattice, together with an algorithm for the upper bound. This operation is reduced to a classical problem of maximum flow in a network. However, we show that the result is sometimes not the most precise we could get, mainly because of the separation between variable equalities and multiset equations. So, this solution is not completely satisfactory, but still deserves to be presented as an unexpected application of max-flow algorithms in this context.

Another solution is presented in Section 6, where variables equalities are considered as (singleton-) multiset equalities, and merged with multiset equations. Now, we have to deal with systems of multiset equations, which are all linear equations. The new idea is to use the classical lattice proposed by Karr as early as 1976 [Kar76], to deal with affine equalities among numerical variables. Since this lattice only uses the affine structure of the space, its operations can be straightforwardly applied to our problem, and provide a well-defined and precise upper bound operator. This solution has been implemented, and some experimental results are given in Section 7.

2 Definitions and notations

For simplicity, we consider a unique set X of *contents values*. As said before, the content of an array should be considered as a multiset of values of X. Since a multiset may contain several instances of the same value, it can be formalized as a function from X to the set of naturals $\mathbb{N}$: if $\mathcal{M}$ is a multiset, $\mathcal{M}(x)$ is

[1] knowing that $i = j$ or $i \neq j$ is essential to know whether an assignment to $A[i]$ may or must affect the value of $A[j]$.

the number of instances of the value x in $\mathcal{M}$. $\mathcal{M}$ is included in $\mathcal{M}'$ (noted $\mathcal{M} \subseteq \mathcal{M}'$ as usual) iff $\forall x \in X, \mathcal{M}(x) \leq \mathcal{M}'(x)$. We use the generalized notion, sometimes called *hybrid multiset* [Syr01,SIYJ07], of multiset with positive or negative multiplicities, i.e., functions from X to the set $\mathbb{Z}$ of integers. $\oplus$ and $\ominus$ denote the sum (disjoint union) and difference of (hybrid) multisets:

$$\mathcal{M} \oplus \mathcal{M}' = \lambda x.\mathcal{M}(x) + \mathcal{M}'(x) \ , \quad \mathcal{M} \ominus \mathcal{M}' = \lambda x.\mathcal{M}(x) - \mathcal{M}'(x)$$

A sum of k instances of $\mathcal{M}$ can be noted $k \otimes \mathcal{M}$.

For simplicity, we restrict ourselves to programs containing integer variables (noted i, j), variables with values in X (noted v, w), and one-dimensional arrays with values in X (noted A, B).

Let A be a one-dimensional array, indexed from 1 to $|A|$ (the size of A). Then $\hat{A}$ denotes the multiset of contents of A, i.e.,

$$\hat{A} = \lambda x. \sum_{i=1}^{|A|} \delta_{x,A[i]} \ , \qquad \text{where } \delta_{x,y} = (\mathit{if}\ x = y\ \mathit{then}\ 1\ \mathit{else}\ 0)$$

An *atom* is either a value in X, or a variable valued in X, or an array cell. If a is an atom, a will also denote the singleton multiset $\lambda x.\delta_{x,a}$.

Our analysis relies on (approximate but conservative) results obtained by two other standard analyzes:

- We use equalities and disequalities about variables and constants used as array indices, to simplify the treatment of aliases: the knowledge that $i = j$ (resp., $i \neq j$) involves that $A[i]$ and $A[j]$ are (resp., are not) aliased. So, we assume the availability of a standard analysis (e.g., based on potentials [Dil89,ACD93], octagons [Min01], or dDBMs [PH07]) giving this kind of informations at each control point. We call it *index analysis*.
- We also use equalities among content variables and array cells. Some of these relations result from the index analysis, others come from assignments and conditional statements. This analysis is called *content analysis*.

So, our abstract values are triples made of

- a system of equations and disequations between integer variables and constants used as array indices, provided by the index analysis;
- a system of equations between atoms, provided by the content analysis;
- a system of equations between multiset expressions, which is computed by our specific *multiset analysis*.

3 An example of analysis

Let's consider the program fragment of Fig. 1.a, which switches the values of two array cells. An analysis could run as follows:

- *At control point 1*, the analysis starts with the multiset equation ($\hat{A} = \mathcal{M}$) — i.e., naming $\mathcal{M}$ the initial content of the array.
- *At point 2*, the content analysis provides the equation ($v = A[i]$), while the previous multiset equation ($\hat{A} = \mathcal{M}$) is preserved.
- *At point 3*, we have to compute the effect of the assignment A[i] := A[j]; two cases may occur:

1. either $i = j$, in which case $A[i]$ and $A[j]$ are aliased, the assignment does nothing, and we get the postcondition $(i = j) \wedge (v = A[i] = A[j]) \wedge (\hat{A} = \mathcal{M})$;
2. or $i \neq j$, and the usual semantics of the assignment provides $(i \neq j) \wedge (A[i] = A[j]) \wedge (\exists x_0, (v = x_0), (\hat{A} = \mathcal{M} \ominus x_0 \oplus A[j]))$ (x_0 is the previous value of $A[i]$, which is overwritten and disappears from the array content, while the value of $A[j]$ is duplicated). So, after simplification, we get $(i \neq j) \wedge (A[i] = A[j]) \wedge (\hat{A} = \mathcal{M} \ominus v \oplus A[j])$

	$\ldots\ldots\ldots \{(\hat{A} = \mathcal{M})\}$
1 v := A[i] ;	$\ldots\ldots\ldots \{(v = A[i]) \wedge (\hat{A} = \mathcal{M})\}$
2 A[i] := A[j] ;	$\ldots\ldots\ldots \{(A[i] = A[j]) \wedge (\hat{A} = \mathcal{M} \ominus v \oplus A[j])\}$
3 A[j] := v ;	$\ldots\ldots\ldots \{(\hat{A} = \mathcal{M})\}$
4	

(a) Program (b) Results

Fig. 1. Switch example

Now, as a postcondition of the assignment, we want to compute an upper approximation of the disjunction

$$\Big((i = j) \wedge (v = A[i] = A[j]) \wedge (\hat{A} = \mathcal{M})\Big) \\ \vee \Big((i \neq j) \wedge (A[i] = A[j]) \wedge (\hat{A} = \mathcal{M} \ominus v \oplus A[j])\Big)$$

This (least) upper bound computation will be the main topic of the paper. Obviously, since the first term of the disjunction contains the equation $(v = A[j])$, it can be rewritten into $(i = j) \wedge (v = A[i] = A[j]) \wedge (\hat{A} = \mathcal{M} \ominus v \oplus A[j])$. Now, both terms contain the same multiset equation and can be unified into $(A[i] = A[j]) \wedge (\hat{A} = \mathcal{M} \ominus v \oplus A[j])$, which is a correct (and precise) postcondition.

– *At point 4*, the computation of the effect of the assignment $\mathsf{A[j]} := \mathsf{v}$ is similar:
  1. either $i = j$, and we get $(i = j) \wedge (A[i] = A[j]) \wedge (\exists x_0, \hat{A} = \mathcal{M} \ominus v \oplus x_0 \ominus x_0 \oplus v)$, i.e., $(i = j) \wedge (A[i] = A[j]) \wedge (\hat{A} = \mathcal{M})$;
  2. or $i \neq j$, and we get $(i \neq j) \wedge (\exists x_0, (A[i] = x_0) \wedge (\hat{A} = \mathcal{M} \ominus v \oplus x_0 \ominus x_0 \oplus v)$, i.e., $(i \neq j) \wedge (\hat{A} = \mathcal{M})$.

  So, the two cases unify into $(\hat{A} = \mathcal{M})$, as expected.

4 Principles of the analysis

As said before, our abstract values are triples $(\varphi_I, \varphi_X, \varphi_\mathcal{M})$, where

– φ_I is a system of equations, and possibly disequations, between indices; it belongs to an abstract lattice $(L_I, \sqsubseteq_I, \sqcup_I, \sqcap_I, \top_I, \bot_I)$.
– φ_X is a system of equations between atoms; it belongs to an abstract lattice $(L_X, \sqsubseteq_X, \sqcup_X, \sqcap_X, \top_X, \bot_X)$.
– $\varphi_\mathcal{M}$ is a system of equations between multiset expressions.

All array cells appearing in φ_X or (as singletons) in $\varphi_\mathcal{M}$ are of the form $A[i]$, meaning that, e.g., $A[i + 1]$ is rewritten as $A[k]$ where k is a fresh variable and the equation $(k = i + 1)$ is expressed in φ_I.

We assume that we analyze a procedure taking arrays $A_1, \ldots, A_p$ as reference parameters, and whose body is made of assignments, conditional statements and loops.

At the entry point of the procedure, a multiset equation is generated for each array, to record its initial content. So the abstract value at the entry point is $\varphi_I = \top_I,\ \varphi_X = \top_X,\ \varphi_{\mathcal{M}} = (\hat{A}_1 = \mathcal{A}_1, \ldots, \hat{A}_p = \mathcal{A}_p)$. Of course, initial knowledge about indices and contents could be taken into account in φ_I and φ_X, instead of taking $\top_I$ and $\top_X$.

We assume that operations are available to propagate abstract values in L_I and L_X among statements, together with widening or acceleration operators to avoid infinite iterations around loops. So, we concentrate on the propagation of multiset equations. Apart from the upper bound — that will be addressed in next sections —, the only non-trivial operation is the assignment to an array cell: let $\boxed{\text{A[i]:=e}}(\varphi_I, \varphi_X, \varphi_{\mathcal{M}})$ denote the effect of the assignment $A[i] := e$ to the abstract value $(\varphi_I, \varphi_X, \varphi_{\mathcal{M}})$. Let J be the set of index variables such that $A[j]$ appears in φ_X or $\varphi_{\mathcal{M}}$. To get the correct and most precise result, we have to consider all the alias cases that should be taken into account, i.e., all the cases where i is equal to some variables $j \in J$. An alias case is subsumed by a subset K of J, interpreted as the index formula $(\forall j \in K, i = j) \wedge (\forall j \in J \setminus K, i \neq j)$. We note $E[\![x/y]\!]$ the substitution of x in place of all occurrences of y in E, and $E[\![x/A[i,K]]\!]$ the substitution in E of x in place of $A[i]$ and all occurrences of $A[j]$, for all $j \in K$. With these notations, the rule of array assignment is the following:

$$\boxed{\text{A[i]:= e}}(\varphi_I, \varphi_X, \varphi_{\mathcal{M}}) = \bigsqcup_{K \subseteq J} \Phi_K$$

where $\Phi_K =$

$$\Big(\varphi_I \sqcap_I \big(\textstyle\bigwedge_{j \in K} j = i\big) \sqcap_I \big(\textstyle\bigwedge_{j \in J \setminus K} j \neq i\big), \tag{1}$$

$$\exists x_0, \tag{2}$$

$$\varphi_X[\![x_0/A[i,K]]\!] \sqcap_X \Big(A[i] = e[\![x_0/A[i,K]]\!] \wedge \bigwedge_{\substack{j \in K \\ \ell = 1..p}} A_\ell[j] = A_\ell[i]\Big), \tag{3}$$

$$\varphi_{\mathcal{M}}[\![x_0/A[i,K]]\!][\![\hat{A} \oplus x_0 \ominus A[i]/\hat{A}]\!]\Big)$$

Each Φ_K is a triple $(\varphi_{I,K}, \varphi_{X,K}, \varphi_{\mathcal{M},K})$ corresponding to an alias case K. In the formula above, line (1) defines $\varphi_{I,K}$ and expresses that K is an alias case (notice that it is $\bot_I$ if φ_I makes it unfeasible), lines (2) and (3) classically involve a common quantified variable x_0 representing the previous value of $A[i]$: line (2) defines $\varphi_{X,K}$ and expresses the changes in φ_X, taking into account the aliases, and line (3) reflects in $\phi_{\mathcal{M}}$ that x_0 represents the common previous value of all the array elements aliased with $A[i]$ and that, in the multiset $\hat{A}$, the previous value of $A[i]$ has been replaced by its new value. Once again, the only non-trivial operation is the upper bound $\sqcup$ that we consider now.

5 Upper bound: a solution based on flows

While the least upper bound operators for systems of index and atom equations are provided by the corresponding lattices, we have to define it for multiset

equations, i.e., to unify two multiset equations, each of which being considered together with a system of atom equations.

Coming back to the computation made at point 3 in the example of §3, we had to unify the multiset equation $\hat{A} \ominus \mathcal{M} = \emptyset$, knowing that $(v = A[i] = A[j])$, with the equation $\hat{A} \ominus \mathcal{M} = A[j] \ominus v$, knowing that $(A[i] = A[j])$.

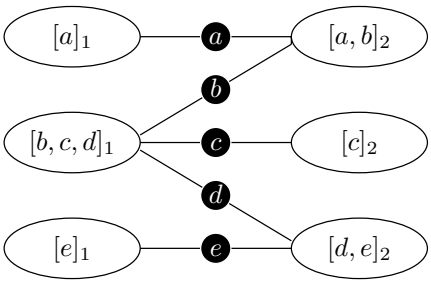
Formally, given two multiset expressions E_1 and E_2 ($\emptyset$ and $A[j] \ominus v$ in our example), and two equivalence relations $\equiv_1$ and $\equiv_2$ over atoms involved in E_1 and E_2 ($(v \equiv_1 A[i] \equiv_1 A[j])$ and $(A[i] \equiv_2 A[j])$ in our example), we want to rewrite each E_i into a common multiset expression E, such that $E_i \equiv_i E, i = 1, 2$.

Basically, the rewriting of an expression E_i into an expression E'_i can be done by adding to E_i a term $a \ominus b$ such that $a \equiv_i b$. In our example, such a rewriting is immediate, since $E_1 = \emptyset \equiv_1 (A[j] \ominus v)$ and $(A[j] \ominus v) = E_2$.

5.1 Rewriting atoms

As said before, when an atom a appears only in E_1, one it can be introduced in E_2 by adding $a \ominus b$ to E_2, for some b such that $a \equiv_2 b$. This rewriting introduces b in E_2, so it may have to be introduced in E_1 in turn, and so on. The process of finding a common rewriting for atoms can be seen as a travel in a graph: consider equivalence classes of $\equiv_1$ and $\equiv_2$ as vertices, and connect two vertices V, V' with an edge labelled by x if $x \in V \cap V'$. The obtained graph is obviously bipartite: each edge connects classes of $\equiv_1$ and $\equiv_2$. The graph drawn below corresponds to an example where we have to unify $E_1 = a$ with $E_2 = e$, knowing $(b \equiv_1 c \equiv_1 d)$ and $(a \equiv_2 b, d \equiv_2 e)$. Now, finding a rewriting from $E_1 = x$ into $E_2 = y$ boils down to finding a path from $[x]_1$ (the vertex corresponding to the class of x according to $\equiv_1$) to $[y]_2$ in the graph: each succession of two edges $\xrightarrow{z}[.]_1\xrightarrow{w}$ around a vertex of class 1 in such a path, corresponds to a rewriting of E_1 into $E_1 \oplus w \ominus z$; conversely, each succession of edges $\xrightarrow{z}[.]_2\xrightarrow{w}$ corresponds to a rewriting of E_2 into $E_2 \oplus z \ominus w$.

For our example (see the opposite figure) the solution is the path $[a]_1 \xrightarrow{a} [a,b]_2 \xrightarrow{b} [b,c,d]_1 \xrightarrow{d} [d,e]_2$: traversing the vertex $[a,b]_2$ corresponds to the rewriting $E_2 = e \equiv_2 a \ominus b \oplus e$, then traversing the vertex $[b,c,d]_1$ corresponds to $E_1 = a \equiv_1 a \ominus b \oplus d$, and finally reaching $[d,e]_2$ allows to deduce $E_2 \equiv_2 a \ominus b \oplus d$, the common rewriting.



5.2 General case

In general, we want to find, if it exists, a common rewriting of multiset expressions E_1 and E_2, which are sums and differences of atoms, possibly with positive coefficients (e.g., $E_1 = a \ominus (2 \otimes b) \oplus (3 \otimes c)$). We could use the previous procedure

to find a common rewriting between each atom in E_1 and an atom of E_2. Instead, we can directly convert our problem into the classical problem of finding a maximal flow in a graph with capacities: we split each expression E_i into a difference $F_i \ominus G_i$ where F_i and G_i are sums of atoms with positive coefficients. Now, we consider again the graph of equivalence classes, as before, where all edges are assigned an infinite capacity, and we extend it with

- a source vertex, labelled with $F_1 \oplus G_2$; for each term $k \otimes a$ in F_1 (resp., in G_2), we create an edge of capacity k from the source vertex to the vertex $[a]_1$ (resp., $[a]_2$);
- and a target vertex, labelled with $F_2 \oplus G_1$; for each term $k \otimes a$ in F_2 (resp., in G_1), we create an edge of capacity k from the vertex $[a]_2$ (resp., $[a]_1$) to the target vertex.

Let's recall that a flow in such a graph with capacities consists of an orientation of the graph (from the source to the target), together with a function ϕ associating with each edge $e = (V_1, V_2)$ of the graph a natural $\phi(e)$, such that (1) for each edge e, $\phi(e)$ does not exceed the capacity $\kappa(e)$ of e, and (2) for each vertex V which is neither the source nor the target, the sum of the values of the flow over all incoming edges to V is equal to the sum of the flow over all outgoing edges. We compute a maximal flow $\phi_{\max}$ from the source to the target (using, e.g, Ford-Fulkerson [FF56] or Edmonds-Karp [EK72] algorithms); if this maximal flow saturates the capacity of edges from the source and to the target, it corresponds to a solution to the initial problem. The common rewriting of E_1 and E_2 is

$$E = \bigoplus_{\substack{V_1 \in \equiv_1 \\ e = V_1 \xrightarrow{a} V_2}} \phi_{\max}(e) \otimes a \;\ominus \bigoplus_{\substack{V_2 \in \equiv_2 \\ e = V_2 \xrightarrow{a} V_1}} \phi_{\max}(e) \otimes a$$

Example: Let $E_1 = a \ominus (2 \otimes b)$, $E_2 = (2 \otimes c) \ominus (3 \otimes d)$, and $(a \equiv_1 c, b \equiv_1 d), (a \equiv_2 d, c \equiv_2 b)$. The corresponding graph is represented by Fig. 2, together with a maximum flow (each edge e is associated with $\kappa(e)/\phi_{\max}(e)$). The common rewriting is then $E = (2 \otimes c) \ominus a \ominus (2 \otimes d)$

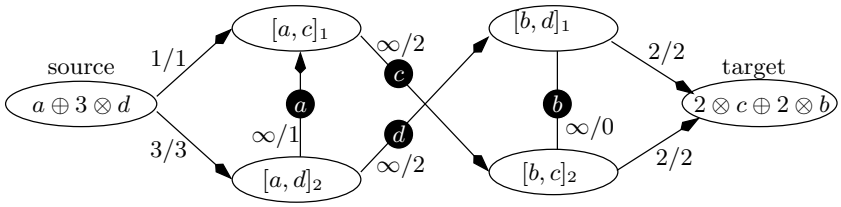


Fig. 2. Max-flow example

Conversely if there is a solution E to the initial problem, it corresponds to a flow : for each atom e in E add one flow unit to the vertex labeled by e, saturate the edges from the source and to the target and the flow conservation will follow from the equation $E_1 \equiv_1 E \equiv_2 E_2$. The reduction is hence sound and complete.

5.3 The canonicity problem

Unfortunately, there can be several maximum flows in the graph, which provide several, non-equivalent solutions for the unified multiset equation. Let's consider the following program fragment:

```
1  d:= A[i]; A[i]:= c;
2  if cond then a:= c; b :=d;
3  else a:= d; b:= c;
4  endif
5  c:= x;
```

Starting with $\hat{A} \ominus \mathcal{M} = \emptyset$ as before, at point 2 we get $\hat{A} \ominus \mathcal{M} = c \ominus d$, then, after the "then" branch, $(a = c \wedge b = d) \wedge (\hat{A} \ominus \mathcal{M} = c \ominus d)$, and after the "else" branch, $(a = d \wedge b = c) \wedge (\hat{A} \ominus \mathcal{M} = c \ominus d)$. After line 4, we have to compute the unification of the previous two values. The max-flow algorithm provides two solutions:

1. $(\hat{A} \ominus \mathcal{M} = c \ominus d)$
2. $(\hat{A} \ominus \mathcal{M} = a \oplus b \ominus (2 \otimes d))$

If we choose the solution (1), after line 5 we lose all information since the value of c is lost; however, if we choose the solution (2) which does not involve c, the equation $(\hat{A} \ominus \mathcal{M} = a \oplus b \ominus (2 \otimes d))$ is preserved after line 5.

So, the max-flow method provides a correct upper bound which is not always the most precise. However, it suggests another approach: notice that the multiplicity of solutions comes from the presence of cycles in the graph. Since the capacities of the edges in a cycle are infinite, we can add or remove flow along a cycle and still get a solution. So, as soon as there is a cycle in the graph, we get an infinite set of solutions. Conversely if there are several solutions, one can be found from another by adding or removing flow along a cycle. An idea is to keep track of the informations given by these cycles. In the previous example before the final assignment, we have the property $a \oplus b = c \oplus d$ (which means $(a = c \wedge b = d) \vee (a = d \wedge b = c)$. It is true since the only difference between the two branches is that c and d are swapped which does not alter the equation. Moreover, this equation corresponds to the cycle found in the graph (which is the same as Fig 2 with different capacities). Adding this equation to one solution is exactly the same thing as increasing the flow along the cycle a, c, b, d. Thus keeping this additional equation between singletons allows us to find any other solution from a single one. The next section describe a more general solution to compute upper bounds which improves the precision since it is able to retain singleton equations and then find the least upper bound.

6 A solution based on linear algebra

Another solution is to consider the atom equations as (singleton) multiset equations, and to handle them together with other multiset equations: then, we get the conjunction of two systems of multiset equations, which are all *linear*. The next idea is to use classical operators in linear algebra, and in particular those

proposed by Karr [Kar76] for propagating affine equations between numerical variables. The key operator is the least upper bound, which computes the system of equations of the least affine space containing two given affine spaces. Of course, we don't consider numerical affine spaces, but the only operations used are those of a vectorial space. Notice also that our equations are always linear (with a null constant term).

The lattice of linear equations: Let us briefly recall the principles of Karr's analysis. Karr's domain is the lattice of linear varieties of an n-dimensional vectorial space. Such a linear variety is defined by a vectorial equation $MX = C$, where M is an $(m \times n)$-matrix, and C is a constant vector. In our special case, the coefficients in M are integers, and C is the null vector (all its components are the empty multiset). There is a classical normal form to this kind of equations, by putting M in *row-echelon form*, using Gauss procedure. The propagation of system of equations through linear assignments is straightforward. The least upper bound operator provides the least linear variety containing its arguments; it is therefore well-defined, geometrically. The procedure proposed by M. Karr is recalled in the appendix: taking 2 matrices M_1 and M_2 in canonical form, it returns a matrix M, also in canonical form, such that the variety $MX = 0$ is the least variety containing both $M_1X = 0$ and $M_2X = 0$.

Coming back to the example of §5.3, we have to compute the least upper bound $\left(a = c \wedge b = d \wedge \hat{A} \ominus \mathcal{M} = c \ominus d\right) \sqcup \left(a = d \wedge b = c \wedge \hat{A} \ominus \mathcal{M} = c \ominus d\right)$ Karr's operator provides the result $a \oplus b = c \oplus d \wedge \hat{A} \ominus \mathcal{M} = c \ominus d$. Eliminating c to compute the result after line 5, we get the precise result $\hat{A} \ominus \mathcal{M} = a \oplus b \ominus (2 \otimes d)$.

Taking into account that the theoretical complexity of the max-flow algorithm (n^2) is better than the complexity of Karr's affine hull (n^3), we could use the max-flow solution to deal with the multiset equations, and apply the affine hull to unify atom equations. However, it is not likely that, in practice, the considered systems of equation become very large, so the complexity is not really an issue.

Let's recall that Karr's lattice is of finite depth: the size of strictly increasing chains is bounded by the number of variables (the dimension of the space), so there is no need for widening to ensure the termination.

Atoms are not numbers

The fact that, in contrast with [Kar76], we are not working in a numerical vectorial space may raise some questions, concerning the existence of solutions, and some implicit consequences of atom equations.

The emptiness problem: some systems of equations have solutions in the usual numerical space, but not in our multiset space. For instance, the equation $a \oplus b = c$, where a, b, c are atoms, has no solution. It is the case of all atom equations which are not balanced (i.e., where the sum of coefficients of both members are not equal). However, this question is not relevant for our analysis, since all the equations considered in the analysis are well-balanced.

Implicit equations: on the other hand, some systems of equations have implicit consequences in our multiset space, which would not occur in the numerical space. For instance, the equation $a \oplus b = 2 \otimes c$, where a, b, c are atoms, implies $a = b = c$. Detecting all such implicit equations is NP-hard (see, e.g., [DV99,DPR08]). However, we don't have any example of program where this kind of implicit equations would appear and be useful. Moreover, finding equalities between atoms is not the goal of the multiset analysis.

So, while algorithms exist for discovering such implicit equations, they were not implemented in our analyzer, because of their excessive cost and the debatable interest of such additional properties. However, notice that this does not change the correctness of our analysis.

7 Experimental results

This analysis has been implemented within the analyzer developed by M. Péron [HP08], taking as input the same language restricted to simple loops and one-dimensional arrays. However, many language restrictions could be easily released for our analysis.

In this section, we show some examples of analyzes. Notice that, for all the examples presented below, the analysis based on flows gives the same results as Karr's lattice. The reported execution times are those with Karr's lattice.

7.1 Insertion sort

We detail below the analysis of the "insertion sort" procedure. We indicate at each line the three properties, respectively concerning indices, contents, and multisets. We assume that the lattice of index properties is the lattice of potentials, with a reasonable widening. The analysis terminates after 3 iterations.

First iteration
for i:= 1 to n do $\dots \{(i = 1), (\hat{A} = \mathcal{M})\}$
  x:=A[i]; j:=i-1; $\dots \{(i = 1, j = i - 1), (x = A[i]), (\hat{A} = \mathcal{M})\}$
  while j>=1 and A[j]>k do $\dots \{(i = 1, j = i - 1), (x = A[i]), (\hat{A} = \mathcal{M})\}$
    A[j+1]:= A[j];$\dots \{(i = 1, j = i - 1), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j])\}$
    j:=j-1; $\dots \{(i = 1, j = i - 2), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j + 1])\}$
  end
  A[j+1] := x;
end

Second iteration
for i:= 1 to n do $\dots \{(i = 1), (\hat{A} = \mathcal{M})\}$
  x:=A[i]; j:=i-1; $\dots \{(i = 1, j = i - 1), (x = A[i]), (\hat{A} = \mathcal{M})\}$
  while j>=1 and A[j]>k do $..(i = 1, 1 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j + 1])$
    A[j+1]:= A[j]; $\dots \{(i = 1, 0 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j])\}$
    j:=j-1; $\dots \{(i = 1, 0 \leq j < i - 1), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j + 1])\}$
  end $\dots \{(i = 1, 0 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j + 1])\}$
  A[j+1] := x;$\dots \{(i = 1, 0 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M})\}$
end

Third iteration
for i:= 1 to n do $\ldots \{(1 \leq i \leq n), (\hat{A} = \mathcal{M})\}$
  x:=A[i]; j:=i-1; $\ldots \{(1 \leq i \leq n, j = i-1), (x = A[i]), (\hat{A} = \mathcal{M})\}$
  while j>=1 and A[j]>k do..$(1 \leq i \leq n, 1 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j+1])$
    A[j+1]:= A[j]; $\ldots \{(1 \leq i \leq n, 1 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j])\}$
    j:=j-1; $\ldots \{(1 \leq i \leq n, 0 \leq j < i-1), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j+1])\}$
  end $\ldots \{(1 \leq i \leq n, 0 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M} \ominus x \oplus A[j+1])\}$
  A[j+1] := x;$\ldots \{(1 \leq i \leq n, 0 \leq j < i), (x = A[i]), (\hat{A} = \mathcal{M})\}$
end $\ldots \{(\hat{A} = \mathcal{M})\}$

7.2 An aliasing surprise

As another simple example, consider two versions of a procedure, intended to perform a circular permutation of the contents of three array cells:

$\ldots \{\hat{A} = \mathcal{M}\}$
x:=A[i]; $\ldots \{x = A[i], \hat{A} = \mathcal{M}\}$
A[i]:=A[j];
$\ldots \{A[i] = A[j], \hat{A} = \mathcal{M} \ominus x \oplus A[j]\}$
A[j]:=A[k];
$\ldots \{A[j] = A[k], \hat{A} = \mathcal{M} \ominus x \oplus A[k]\}$
A[k]:=x; $\ldots \{A[k] = x, \hat{A} = \mathcal{M}\}$

(a) rotation

$\ldots \{\hat{A} = \mathcal{M}\}$
x:=A[i];$\ldots \{x = A[i], \hat{A} = \mathcal{M}\}$
y:=A[j];$\ldots \{x = A[i], y = A[j], \hat{A} = \mathcal{M}\}$
z:=A[k];
$\ldots \{x = A[i], y = A[j], z = A[k], \hat{A} = \mathcal{M}\}$
A[i]:=y;$\ldots \{y = A[j] = A[i], \hat{A} = \mathcal{M} \ominus x \oplus y\}$
A[j]:=z;$\ldots \{A[j] = z, \hat{A} = \mathcal{M} \ominus x \oplus z\}$
A[k]:=x; $\ldots \{A[k] = x\}$

(b) copy-store

Fig. 3. Permuting 3 values

- The first version (Fig. 3.a) performs a simple rotation, using a buffer x. The analysis proves that the final content of the array is a permutation of the initial one.
- In the second version (Fig. 3.b), the three values are first copied in buffers, and then stored back at their respective places. On that program, the analysis is not able to show anything interesting about the final content. Of course, this could result from some imprecision; but if we look closer at the cause of the failure, it appears that there is a case where the content of the array is not preserved: if $i = k \neq j$, the initial value of $A[i]$ is copied twice in the final array, and the initial value of $A[j]$ is lost. So, our analysis is precise and detects a bug in the program.

7.3 Combining the analysis with array partitioning

Our analysis can be easily combined with the methods [GRS05,HP08] which partition the arrays into symbolic slices, and associate a *summary variable* with each such slice.

For instance, we used the method of [HP08] to partition arrays into relevant slices, and used our abstract domain to analyze the properties of these slices.

Program	exp. res.	results	time	nb.iter.
switch 2 cells by rotation	$\hat{A} = \mathcal{A}$	ok	<4ms	1
switch 2 cells by copy-store	$\hat{A} = \mathcal{A}$	ok	<4ms	1
switch 3 cells by rotation	$\hat{A} = \mathcal{A}$	ok	<4ms	1
switch 3 cells by copy-store	$\hat{A} \neq \mathcal{A}$	ok	<4ms	1
Dutch national flag [Dij76]	$\hat{A} = \mathcal{A}$	ok	<4ms	2
Insertion sort	$\hat{A} = \mathcal{A}$	ok	<4ms	3
Selection sort	$\hat{A} = \mathcal{A}$	ok	<4ms	2
Bubble sort	$\hat{A} = \mathcal{A}$	ok	<4ms	3
With array partitioning				
Array copy	$A\widehat{[0..n-1]} = B\widehat{[0..n-1]}$	ok	< 4ms	2
Split on sign [KV09]	$A\widehat{[0..a-1]} = B\widehat{[0..b-1]} \oplus C\widehat{[0..c-1]}$	ok	12ms	2

Table 1. Some experimental results

In [HP08], array accesses and loop indices are used to separate array cells that should be considered separately (as singleton slices). Then an array is partitioned into these singleton slices, and contiguous slices separating these singletons.

Only a small amount of work is needed to adapt the abstract domain to this slicing technique. When an index is progressing, a slice may be growing and then we have to perform a substitution in the equation system to reflect that the new slice is the union of old slice with some singleton.

This combination of techniques can now be used to find some other interesting properties. For instance in a program which copies an array A to an array B we are now able to state at each step of the loop indexed by i that the multisets of values of cells with index greater than 0 but less than i are equals in the array A and in the array B. Then this intermediate property allows us to discover the multiset equality of A and B and finally use it to prove more specific properties.

The opposite program is another interesting example considered in [KV09]: it splits an array A into B and C according to the signs of the elements. Using our combined analysis we get respectively for each array A, B and C the partitions $\{A[0..a-1], A[a], A[a+1..n-1]\}$, $\{B[0..b-1], B[b], B[b+1..n-1]\}$ and $\{C[0..c-1], C[c], B[c+1..n-1]\}$. Propagating the multiset properties between these slices, we find the expected loop invariant:

$$A\widehat{[0..a-1]} = B\widehat{[0..b-1]} \oplus C\widehat{[0..c-1]}$$

which, once again, could not be expressed as a first-order formula.

```
a := 0, b := 0, c := 0 ;
while a < n do
    if A[a] ≥ 0 then
        B[b] := A[a] ;
        b++ ;
    else
        C[c] := A[a] ;
        c++ ;
    a++ ;
```

7.4 Other examples

Table 1 shows the analysis time for several small programs. All the results are as expected.

7.5 An example with linked data structures

Our lattice of multiset equations can be used for other data structures than arrays. As an example of possible application, we have analyzed (by hand) the Deutsch-Schorr-Waite data structure traversal algorithm [SW67]. In fact, we consider the version of [Lin73], dedicated to data structures without cycles, and which does not involve any auxiliary marking. A slightly different version of this algorithm has been completely proven with TVLA [LRS06]. We recall that this algorithm traverses a binary structure (a dag or a tree), without using a stack, by redirecting pointers in the structure to store the return path. We note $\hat{T}$ the multiset of pointers contained in the structure, i.e., contained in a node initially reachable from the root of the structure. The goal would be to show that this multiset is restored at the end of the traversal (of course, this does not show that the structure has been restored). The results are shown in Fig. 4. The goal is not reached, because it needs the additional fact that, at the end of the program, **root = prev**, a fact that would need another kind of analysis, and the knowledge that "**-1**" does not appear in the initial structure. However, the computed invariants are precise.

prev:=-1; cur:=root; $\ldots \{prev = -1, cur = root, \hat{T} = \mathcal{M}\}$
while cur<>-1 $\ldots \{\hat{T} = \mathcal{M} \oplus root \oplus -1 \ominus cur \ominus prev\}$
  next:=cur->left; cur->left:=cur->right;
  cur->right:=prev; $\ldots \{\hat{T} = \mathcal{M} \oplus root \oplus -1 \ominus cur \ominus next\}$
  prev:=cur; cur:=next; $\ldots \{cur = next, \hat{T} = \mathcal{M} \oplus root \oplus -1 \ominus prev \ominus next\}$
  if cur=NULL $\ldots \{cur = next = NULL, \hat{T} = \mathcal{M} \oplus root \oplus -1 \ominus prev \ominus next\}$
    cur:=prev; prev=NULL;
    $\ldots \{prev = next = NULL, \hat{T} = \mathcal{M} \oplus root \oplus -1 \ominus cur \ominus next\}$
  end $\ldots \{\hat{T} = \mathcal{M} \oplus root \oplus -1 \ominus cur \ominus prev\}$
end $\ldots \{\hat{T} = \mathcal{M} \oplus root \ominus prev\}$

Fig. 4. Results for the Deutsch-Schorr-Waite algorithm

8 Conclusion

To our knowledge, it is the first automatic analysis for handling permutation-invariant properties of data-structures. Basically, our abstract values are equations between multiset expressions, together with equations, gathered by other analyzes, between locations (indices, pointers) and structure contents. Two ways for computing least upper bounds of multiset equations have been proposed: the solution based on flows is theoretically more efficient, but may be less precise in general; the other solution makes use of the standard lattice of linear equations, and deals jointly with multiset and content equations.

The paper is specialized to the analysis of arrays, but our lattice could be used for any kind of data structures, as shown by the Deutsch-Schorr-Waite example, provided a suitable interpretation of statements on these data-structures is available. This would give a relevant abstraction for every collection data structure and thus could be used in a shape-value abstraction [Vaf09] in conjunction with

a shape analysis to derive properties of these structures or to use these properties in programs manipulating collections. It would be also useful to consider more general programs (e.g., recursive programs) and statements (e.g., indirect indexing), but this would not interfere with the definition of the lattice and its operations. Another perspective is to consider multiset inclusions, in order to be able to show that some data structure is included, up to some permutation, inside another one.

References

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993. Preliminary version appears in the Proc. of 5th LICS, 1990.

[BMS06] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In E. A. Emerson and K. S. Namjoshi, editors, *VMCAI 06*, pages 427–442. LNCS 3855, Springer Verlag, 2006.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976.

[Cou03] P. Cousot. Verification by abstract interpretation. In N. Dershowitz, editor, *Proc. Int. Symp. on Verification – Theory & Practice – Honoring Zohar Manna's 64th Birthday*, pages 243–268, Taormina, Italy, June 29 – July 4 2003. © Springer-Verlag, Berlin, Germany.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation, 1976.

[Dil89] D. L. Dill. Timing assumptions and verification of finite state concurrent systems. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[DPR08] A. Dovier, C. Piazza, and G. Rossi. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM TOCL*, 9(3), May 2008.

[DRS03] N. Dor, M. Rodeh, and M. Sagiv. CCSV: towards a realistic tool for statically detecting all buffer overflows in C. In *ACM Conference on Programming Language Design and Implementation, PLDI 2003*, pages 155–167, San Diego, June 2003.

[DV99] E. Dantsin and A. Voronkov. A nondeterministic polynomial-time unification algorithm for bags, sets and trees. In *FOSSACS'99*, pages 180–196. LNCS 1578, , Springer, 1999.

[EK72] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *JACM*, 19(2):248–264, 1972.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL 2002*, pages 191–202. ACM, 2002.

[GMT08] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In G. C. Necula and P. Wadler, editors, *POPL 2008*, pages 235–246. ACM, 2008.

[GRS05] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *Proc. of POPL'2005*, pages 338 – 350, Long Beach, CA, 2005.

[HP08] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM Conference on Programming Language Design and Implementation, PLDI 2008*, pages 339–348, Tucson (Az.), June 2008.

[IHV08] R. Iosif, P. Habermehl, and T. Vojnar. What else is decidable about arrays? In R. Amadio, editor, *FOSSACS 2008*. LNCS, Springer Verlag, 2008.

[JM07] R. Jhala and K. L. McMillan. Array abstractions from proofs. In W. Damm and H. Hermanns, editors, *CAV 2007*, pages 193–206. LNCS 4590, Springer Verlag, 2007.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[KV09] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE 2009*, pages 470–485. LNCS 5503, Springer, 2009.

[LB04] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In R. Alur and D. Peled, editors, *CAV 2004*, pages 135–147. LNCS 3114, Springer Verlag, 2004.

[Lin73] G. Lindstrom. Scanning list structures without stacks or tag bits. *Information Processing Letters*, 2(2):47–51, June 1973.

[LRS06] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS 2006*, pages 261–279, Seoul, Korea, 2006. LNCS 4134, Springer.

[LS79] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in Pascal. *ACM Trans. Program. Lang. Syst.*, 1(2):226–244, 1979.

[Min01] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[PH07] M. Péron and N. Halbwachs. An abstract domain extending Difference-Bound Matrices with disequality constraints. In B. Cook and A. Podelski, editors, *8th International Conference on Verification, Model-checking, and Abstract Interpretation, VMCAI'07*, Nice, France, January 2007.

[SG09] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *ACM Conference on Programming Language Design and Implementation, PLDI 2009*, pages 223–234, 2009.

[SIYJ07] D. Singh, A.M. Ibrahim, T. Yohanna, and J.N.Singh. An overview of the applications of multisets. *Novi Sad J. Math.*, 37(2):73–92, 2007.

[SJ80] N. Suzuki and D. Jefferson. Verification decidability of Presburger array programs. *JACM*, 27(1), January 1980.

[SRW02] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[SW67] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[Syr01] A. Syropoulos. Mathematics of multisets. In Cristian Calude, Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Multiset Processing, WMP 2000*, volume 2235 of *Lecture Notes in Computer Science*, pages 347–358. Springer, 2001.

[Vaf09] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In N. Jones and M. Müller-Olm, editors, *10th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI'09*, pages 335–348, Savannah (GA), January 2009. LNCS 5403, Springer Verlag.

[WSR00] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In David A. Watt, editor, *Compiler Construction, CC2000*, pages 1–17. LNCS 1781, Springer, 2000.

Appendix: Karr's algorithm for linear hull

Let's recall the algorithm proposed in [Kar76] for computing the linear hull: given two linear subspaces, defined by their matrices M and M' in reduced row-echelon form (i.e., (1) each row has at least one non-zero entry, and for any row i_0, if j_0 is the first column with a non zero entry of the row, then (2) for all $i > i_0, j <= j_0$, $M_{ij} = 0$ and (3) forall $i \neq i_0, M_{i,j_0} = 0$), the algorithm returns the matrix (in reduced echelon form) of the least linear subspace containing the two given subspaces. The algorithm progressively modifies M and M' so that, at the end of step s, the first s columns of M and M' are equal. After n steps, M and M' are equal, and it is the solution. The two matrices are maintained in the following form:

$$M = \left(\begin{array}{c|c} C & N \\ 0 & \end{array}\right) \ , \ M' = \left(\begin{array}{c|c} C & N' \\ 0 & \end{array}\right)$$

where the common part C has s columns at step s.

At the beginning of step s, let r be the number of rows of C plus 1. There are 3 cases according to the values of N_{rs} and N'_{rs}:

1. either $N_{rs} = N'_{rs} = 1$, then, from the hypotheses on M and M', we have:

$$M = \left(\begin{array}{c|c} C & \begin{array}{c}\vdots\\ 0\end{array} \\ \hline & \boxed{1} \end{array} N\right) \ , \ M' = \left(\begin{array}{c|c} C & \begin{array}{c}\vdots\\ 0\end{array} \\ \hline & \boxed{1} \end{array} N'\right)$$

   and we just increment r and s;
2. or $N_{rs} = 1$ and $N'_{rs} = 0$ (or conversely), and the matrices are in the form:

$$M = \left(\begin{array}{c|c} C & \begin{array}{c}\vdots\\ 0\end{array} \\ \hline & \boxed{1} \end{array} N\right) \ , \ M' = \left(\begin{array}{c|c} C & \beta \\ \hline & \end{array} N'\right)$$

   then, M is modified by obtaining the column β in the $r-1$ positions of column s (previously 0), by performing suitable linear combinations of there $r-1$ rows with row s. Row s of M is then suppressed.
3. or $N_{rs} = N'_{rs} = 0$, and the matrices are in the form:

$$M = \left(\begin{array}{c|c} C & \alpha \\ \hline & \end{array} N\right) \ , \ M' = \left(\begin{array}{c|c} C & \beta \\ \hline & \end{array} N'\right)$$

   If columns α and β are the same, s is just incremented. Otherwise, let ℓ be the greatest row index such that $\alpha_\ell \neq \beta_\ell$. Then, in each matrix, let R_ℓ be the row ℓ and R_i be a row on index $< \ell$; replace each R_i by $R_i - (\alpha_i - \beta_i)/(\alpha_\ell - \beta_\ell)R_\ell$; finally, delete row ℓ in both matrices: columns s are the same in both matrices, and s can be incremented.