Array Expansion

Paul Feautrier[*]
Laboratoire PRiSM,
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

July 1988

Abstract

A common problem in restructuring programs for vector or parallel execution is the suppression of false dependencies which originate in the reuse of the same memory cell for unrelated values. The method is simple and well understood in the case of scalars. This paper gives the general solution for the case of arrays. The expansion is done in two steps: first, modify all definitions of the offending array in order to obtain the single assignment property. Then, reconstruct the original data flow by adapting all uses of the array. This is done with the help of a new algorithm for solving parametric integer programs. The technique is quite general and may be used for other purposes, including program checking, collecting array predicates, etc...

1 Introduction

1.1 Motivation

One of the most striking trends in today's computer architecture is the development of special purpose machines for numerical computations. The idea behind this effort is that by capitalizing on the peculiarities of scientific computing, one may improve on the performance-to-price ratio of general purpose processors.

Scientific computation is characterized by its highly repetitive nature, as found for instance in the use of vectors and arrays with regular adressing patterns. Hence numerical super-computers are parallel or pipelined machines with many processing elements among which the total workload is distributed.

[*]e-mail : `Paul.Feautrier@prism.uvsq.fr`

The price to pay for the increased power is a more difficult programming task. When using sequential languages, parallel and vector operations must be translated to lower-level constructions. To make efficient use of a super-computer, one must retrieve these features from the sequential program text or (worse !) from one's sequential thinking.

Many attempts have been made to automate this process by constructing optimizing and restructuring compilers; see [PW86] for an up-to-date review. Broadly speaking, all such compilers start by detecting dependencies in the source code. Two instructions are dependent if the first one compute a value which is used by the other (Producer-Consumer or data dependency), or if the first one use a variable which is modified by the second (Consumer-Producer or anti-data dependency), or if both modify the same variable (Producer-Producer or output dependency). If two instructions are independent, one may execute them in parallel; in the special case where these instructions are instance of the same program text (as happens in a loop), one may pipeline them.

Most often, the dependence relation is summarized by a dependency graph. There are several algorithms to extract a parallel or vector program from this graph, the most comprehensive being probably the one of [AK84]. Whatever the algorithm, it is clear that edges in the dependency graph are obstacles to parallelization. Now, any algorithm is susceptible of many different implementations; it is natural to search for programs which, while giving the same ultimate results, have more opportunities for parallel execution, i.e. less edges in their dependency graph.

When looking for such edge cutting transformations, one sees at once that PC edges are inherent to the structure of the algorithm, while the presence of CP and PP edges indicate merely that a memory cell has be reused. If programs where written in single assignment style, there would be no such edges. For instance, if a scalar variable is modified in a loop, this creates a PP edge which precludes parallel execution. Replacing the scalar by an array whose index is the loop counter will cut the PP edge and may (in favorable cases) allow parallel execution. This is the well-known scalar expansion technique [Wol78]. The aim of this paper is to extend it to the general case, in which one tries to cut a PP edge on an array; hence the name Array Expansion.

1.2 Outline

Paragraph 2 will set the stage for the paper by describing the simple programming language we will work with, and introducing necessary notations and definitions. Paragraph 3 will describe, with the help of an exemple, the two-step expansion process: first, rename or expand arrays definitions as necessary, then restore the correct data flow by rewriting references to the modified arrays. In paragraph 4, we will return to the simpler problem of scalar expansion, and show that classical techniques are particular cases of our general solution. In the conclusion, we will review our results and point to some unsolved problems.

1.3 Related work

1.3.1 Scalar expansion

Scalar expansion (see [PW86] or [Wol78]), is mainly used by vectorizing compilers; the transformation is restricted to innermost loops in which a scalar is assigned to. The situation may be corrected by replacing the scalar by an array whose index is the loop counter:

```
for i := 1 to n do        for i := 1 to n do
begin                     begin
      ...                       ...
      x := ...   ;   ⟹         xx[i] := ...   ;
      ...                       ...
end;                      end;
```

In the loop body, references to `x` must be replaced by `xx[i]` if they occur after the defining instruction, and `xx[i-1]` otherwise. One must also add an initialisation instruction before the loop:

```
xx[0] := x;
```

and restore the contents of `x` after the loop:

```
x := xx[n];
```

There are various schemes (strip mining, use of vector registers, etc) to limit the amount of memory space which is used up by the new array `xx`.

A closely related technique is node splitting, in which a subexpression is equated to a temporary, which is then expanded into an array. In this case, neither initialisation nor restoration is necessary.

1.3.2 Use Def Chains

The computation of use-def chains, (see [ASU86], chapter 11), is a technique which is used in optimizing compilers to summarize the flow of data in a program. Before executing an optimizing transformation (such as code motion or dead code elimination), the information is used to verify that the proposed transformation is valid. To each use (rhs occurence) of a variable `x` is associated a list of definitions of `x` which may be the source of the current value of `x`. Use-def chains are computed by iteratively solving propagation equations.

The theory of use-def chains is both more and less comprehensive than the present one. Its range of applicability is wider, since the source program may contain conditionals and even `goto`'s. However, it is a static theory (all executions of an instruction in a loop are lumped as one), and, as such, apply only to scalars (or to arrays considered as a whole). Whether a synthesis is possible will be the subject of future research.

2 Notations and conventions

We will first describe, with the help of an incomplete BNF grammar, the syntax of our source language. We will then discuss the restrictions we superimpose on this grammar. In some cases, these restrictions may be lifted by more complicated algorithms or by preprocessing. We will then introduce the sequencing predicate, as a mean of finding the execution order of instructions instances.

2.1 The source language

The source language may be seen either as a static PASCAL or a rationalized FORTRAN. Data types will be restricted to integers, reals, and n-dimensional arrays of integers and reals. Declarations, which are of interest to the compiler only, will be omited. We will use without definition the categories of `<identifiers>` and `<simple expressions>`; their syntax will be the classical Pascal one.

2.1.1 Specification

The language will be a subset of Pascal. The only simple instructions we will consider will be scalar and array assigment. The only control constructions will be the sequence and the for loop. We will extend the language in order to allow conditional expressions, which are necessary for the expression of index calculations (see e.g. paragraph 3.2.3). The syntax will be:

```
<conditional expression> ::=
    if <boolean expression> then <expression> else <expression>
```

Note the abscence of `goto`'s, of conditional instructions, of `while` loops and of procedures.

2.1.2 Restrictions

The programs we will consider in the sequel must conform to quite stringent restrictions, beside those inherents in the above specification.

Some restrictions are enforced merely to simplify the exposition. It is often possible to implement preprocessors to eliminate the offending constructs. Other features (e.g. `while` loops or non-linear indices) prohibit the use of the proposed technique.

The central problem we try to solve here is the following: given an array element, which of several instruction is the source of the value contained therein. Most of the time, the defining instructions will be embedded in loops. The values of indices as a function of the loop counter, and the iteration count of the loop are evidently crucial knowledge for the solution. These informations are easy to obtain for normal `for` loops and for affine indices. `while` loops, on the other hand, are quite intractable and will be prohibited.

Normal loops For the reasons given above, we will only consider `for` loops. Furthermore, in the interest of simplicity, the step will always be 1. It is quite easy to implement a preprocessor to reduce all `for` loops to the above format. It is also possible to eliminate `goto`'s (see for instance [Bak77]), to detect induction variables ([ASU86]) and to detect `while` loops which are `for` loops in disguise.

Linear indices All indices will be restricted to affine functions of the loops inductions variables and of other integer variables. Similarly, all loops upper bounds will be affine functions of the surrounding loops induction variables and of other integer variables. These auxilliary variables will be treated as constants throughout this paper. This restriction may sometime be lifted by semantic analysis (see [CH78] or [Jou87]).

The source program is correct! We will use the fact that in a correct program, array indices are always within the array bounds. Hence, two array references address the same memory location if and only if they are references to the same array and if their indices are equal.

Obviously, it is good programming practice to debug a sequential program before attempting to restructure it for parallel or vector execution. Hence the restriction is not too severe.

2.2 The sequencing predicate

Values in array elements are not produced by instructions, but by instructions executions. Hence we need a notation to designate a specific execution of an instruction.

Our first need is an unambiguous designation of an instruction in a program. Neither the text of the instruction nor its position in the program syntax tree will serve, since there may be several instructions with the same text, and since the program may be modified by a restructuring compiler. Hence we will use a set of arbitrary instruction names, which will be denoted by letters such as r, s, etc... In a practical application, a natural choice for these names may be pointers to records containing the instruction descriptions. In the sequel, we will mostly be interested in simple instructions. However, some discussions will be clearer if all instructions, compound or simple, are named.

In our source language, the only repetitive construct is the `for` loop. Hence, an instruction instance is uniquely defined by the name of the instruction and the values of the surrounding loop induction variables (the iteration vector of [Kuc78]). A pair whose components are an instruction name and an integer list will be called an instruction coordinate. To denote an instruction instance, a coordinate must verify two conditions:

- the length of the integer list must be equal to the number of loops surrounding the instruction;
- all integers in the list must be within the corresponding loop limits.

In the following, integer lists will be denoted by **bold** letters. [] will denote the empty list. $\mathbf{a}[i..j]$ will denote the sublist of $\mathbf{a}$ beginning at position i and ending at j, while $\mathbf{a}[\mathbf{i}]$ will be an abbreviation for $\mathbf{a}[\mathbf{i}..\mathbf{i}]$. We will use freely extensions of usual predicates to integer lists and functions whose values are integer lists. For instance,

$$\mathbf{F}(i) \geq 0$$

asserts that function $\mathbf{F}$ return a list whose elements are non negative integers.

To each looping instruction t we may associate a pair of inequalities:

$$lb_t \leq a \leq ub_t,$$

where a is the loop counter of t. If an instruction s is embedded in a loop nest $t_1, t_2, \dots, t_N$, in that order, then the iteration vector of s, $\mathbf{a}$, must satisfy:

$$\forall p : (1 \leq p \leq N)\, lb_{t_p} \leq \mathbf{a}[p] \leq ub_{t_p}. \qquad (1)$$

According to 2.1.2, loop t_p bounds are affine functions of the iteration counters of loops t_j with $j < p$. (1) may be summarized in matrix form:

$$L_s\mathbf{a} + h_s \geq\!> 0. \qquad (2)$$

where L_s is a quasi-triangular matrix.

Obviously, in deciding whether coordinate $(r, \mathbf{a})$ or $(s, \mathbf{b})$ is the source of a given value, we need to know which of $(r, \mathbf{a})$ or $(s, \mathbf{b})$ is executed first. The fact that $(r, \mathbf{a})$ is executed before $(s, \mathbf{b})$ will be expressed by:

$$(r, \mathbf{a}) \prec (s, \mathbf{b}).$$

$\prec$, the sequencing predicate, is a strict total order on the set of coordinates. $\prec$ depends on the source program text. In fact, in applications in which we consider several texts (e.g. an initial sequential text and a parallel one), there should be several sequencing predicates, which must be distinguished by subscripting. Our aim is to give a simple expression for $\prec$.

Suppose first that r and s are instructions in the outermost instruction list of the program. **a** and **b** necessarily are the empty list $[]$. $(r, []) \prec (s, [])$ iff r precedes s in the program text. Let T_{rs} be a boolean such that T_{rs} is true iff r textually precedes s. In this case:

$$(r, []) \prec (s, []) = T_{rs}.$$

Note that $T_{rr} = \mathbf{false}$ and that if $r \neq s$, $T_{rs} = \neg T_{sr}$.

Suppose next that r and s are the same instruction. In this case, according to the usual semantics of for loops, $(r, \mathbf{a}) \prec (r, \mathbf{b})$ iff **a** is lexicographically smaller than **b**. Lexicographic ordering will be denoted by the symbol $\ll$.

In the general case, there is an innermost loop t whose body contains both r and s. Let N_{rs} be the depth of this loop. In the body of t, there are two instructions r' and s' such that r is r' or is inside r', and s is s' or is inside s'. Obviously:

$$(r, \mathbf{a}) \prec (s, \mathbf{b}) \equiv (r', \mathbf{a}[1..N_{rs}]) \prec (s', \mathbf{b}[1..N_{rs}]).$$

Now, if $\mathbf{a}[1..N_{rs}] \neq \mathbf{b}[1..N_{rs}]$, $(r', \mathbf{a})$ and $(s', \mathbf{b})$ belong to distinct iterations of loop t. In this case, their order is given by a lexical comparison of $\mathbf{a}[1..N_{rs}]$ and $\mathbf{b}[1..N_{rs}]$. Conversely, if $\mathbf{a}[1..N_{rs}] = \mathbf{b}[1..N_{rs}]$, then $(r', \mathbf{a})$ and $(s', \mathbf{b})$ belong to the same iteration of t, and their order is the textual order $T_{r's'} \equiv T_{rs}$. Putting all this together:

$$(r, \mathbf{a}) \prec (s, \mathbf{b}) = \mathbf{a}[1..N_{rs}] \ll \mathbf{b}[1..N_{rs}] \vee (\mathbf{a}[1..N_{rs}] = \mathbf{b}[1..N_{rs}] \wedge T_{rs}). \quad (3)$$

Knowledge of N_{rs} (a set of integers) and T_{rs} (a set of booleans) is all we need to sequence all instructions in a program. The definition of the $\prec$ predicate may be extended to programs containing parallel constructions. In this case, $\prec$ is no longer a total order. Similarly, in the presence of well structured conditionals, $\prec$ may still be defined. Since instructions in opposite arms of an `if` have no temporal relationship, $\prec$ is a partial order in this case also. Extending the notation to unstructured programs is apparently impossible, especially since there is no longer a uniform method for specifying a particular instruction instance. This is the gist of the classical arguments of [Dij68] against the use of `goto`'s.

In the balance of the paper, we will use the following example:

```
for k := 0 to m+n do
{r} c[k] = 0. ;
for i := 0 to m do                                  {A}
    for j := 0 to n do
{s}     c[i+j] := c[i+j] + a[i] * b[j];
```

which compute the product of two polynomials `a` and `b` of respective degrees m and n. There are two elementary instructions to which we have given the names r and s. A coordinate for r is of the form $(r, [x])$, where x must satisfy $0 \leq x \leq m + n$. Similarly, a coordinate for s is $(s, [y, z])$, where:

$$\begin{array}{rcl} 0 \leq & y & \leq m, \\ 0 \leq & z & \leq n. \end{array} \tag{4}$$

It is quite clear that $N_{rs} = 0$ and T_{rs} is true; this is another expression of the obvious fact that all instances of r are executed before any instance of s. On the other hand, $N_{ss} = 2$ and T_{ss} is false; hence:

$$(s, [y, z]) \prec (s, [y', z']) \equiv y < y' \vee y = y' \wedge z < z'. \tag{5}$$

Observe that the elements of the coordinate list act as bound variables. We will feel free to rename them (as we have done in (5)) in order to avoid collisions.

3 The expansion process

There is a very strong relation between PP and CP edges. In fact, one may prove that in a correctly written program, if there is a CP dependency on `x` from s to t, there exists another instruction r with a PP dependency from r to t.

In a correct program, each variable must be set before being used. Since s uses (consumes) `x`, there is another instruction r, which is executed before s and sets `x`. Now, "before" is transitive; s is executed before t, and s and t are in PP dependency.

From this we deduce that if all PP dependencies on `x` are deleted, then, as a by-product, CP dependencies will also disappear.

Array expansion is a two step process. First, one selects an array production and rewrites it in such a way that no memory word is written more than once. Next, all uses of the array must be examined to find whether the reference is to the new array or the old; in the first case, new values of the indexing function must be determined.

3.1 Renaming and expanding

Suppose our aim is to cut a PP edge $s \rightarrow t$ of the dependency graph. In our source language, this implies that s and t are both assignment to the same array or scalar `A`. To suppress the dependency, we must insure that, in the modified version of the program, s and t assign values to different memory locations. There are two ways of insuring this property. If s and t are different instructions, simply create a new array `A'` similar to `A`, and replace in one instruction (say t) the left hand side reference to `A` by `A'` (Array Renaming). Obviously, this technique does not work if s and t are the same instruction.

Consider the case of an instruction r with coordinate $(r, \mathbf{a})$:

$$\mathtt{A}[\mathbf{f}(\mathtt{a})] := \ldots; \tag{6}$$

There is a PP dependancy on r iff the following system:

$$\begin{array}{rcl} L_r\mathbf{a} + \mathbf{h}_r & \geq & 0, \\ L_r\mathbf{a}' + \mathbf{h}_r & \geq & 0, \\ \mathbf{f}(\mathbf{a}) & = & \mathbf{f}(\mathbf{a}'), \\ (r, \mathbf{a}) & \prec & (r, \mathbf{a}'), \end{array}$$

has solutions. Now it is clear that if $\mathbf{f}$ is the identity function: $\mathbf{f}(\mathbf{a}) = \mathbf{a}$, there is no solution since $(r, \mathbf{a}) \prec (r, \mathbf{a}')$ is always false. Hence, to suppress the PP edge, rewrite r as :

$$\mathtt{A'}[\mathbf{a}] := \ldots$$

Clearly, the new array `A'` has more entries that `A`, hence the name Array Expansion.

In real cases, the renaming policy may be more complicated. For instance, if the target machine is a pipeline processor, we are interested only in suppressing dependencies in the innermost loop; hence, it is sufficient to expand the lhs of r:

$$\mathtt{A'}[\mathbf{a}[N_{rr}]] := \ldots$$

The study of this and other strategies is left for future research.

In the case of example {A}, there are two PP dependencies, which are due to the fact that both r and s rewrite several times the same cell of `A`. It so happens that, by replacing `c` in s by a new two-dimensional array `cc`, both dependencies are cut. The original program becomes:

```
for k := 0 to m+n do
{r} c[k] = 0.;
for i := 0 to m do                          {B}
    for j := 0 to n do
{s}     cc[i,j] := ? + a[i] * b[j] ;
```

The problem lies in the rewriting of the rhs of s.

3.2 Reconstructing the data flow

3.2.1 Some notations

Suppose that we are given a program conforming to the restrictions of paragraph 2. Let t be an instruction in which an array `A` is used. Let $\mathbf{b}$ be the iteration vector of t; the indices of `A` are affine functions of $\mathbf{b}$. In vector form, the reference to `A` may be written $\mathtt{A}[\mathbf{g}(\mathbf{b})]$.

We are interested in finding the origin of the value of $\mathtt{A}[\mathbf{g}(\mathbf{b})]$. Let $s_1, s_2, \ldots, s_n$ be the instructions wich produce a value for `A`, $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ their iteration vectors. s_i is of the form:

$$\mathtt{A}[\mathbf{f}_i(\mathbf{a}_i)] = \ldots$$

3.2.2 Formal solution

Any instruction s_i may be the source of the value used by t; let us denote by $(s_i, \mathbf{K}_i(\mathbf{b}))$ a possible source for t. The real source is the latest such instruction; it is $(s_i, \mathbf{K}_i(\mathbf{b}))$ iff:

$$\forall j \neq i, (s_j, \mathbf{K}_j(\mathbf{b})) \prec (s_i, \mathbf{K}_i(\mathbf{b})). \tag{7}$$

The correct value of i may depend on $\mathbf{b}$. In particular, $\mathbf{K}_i(\mathbf{b})$ may be undefined for some value of $\mathbf{b}$. We will suppose that an undefined iteration vector (written as ∞) comes earlier than any other coordinate:

$$\forall s, t, \mathbf{b} : (s, \infty) \prec (t, \mathbf{b}). \tag{8}$$

The conditions on $\mathbf{K}_i(\mathbf{b})$ are:

- Firstly, $(s_i, \mathbf{K}_i(\mathbf{b}))$ must produce a value for $\mathtt{A}[\mathbf{g}(\mathbf{b})]$:

$$\mathbf{f}_i(\mathbf{K}_i(\mathbf{b})) = \mathbf{g}(\mathbf{b}) \tag{9}$$

- Secondly, $(s_i, \mathbf{K}_i(\mathbf{b}))$ must be the latest such coordinate:

$$\mathbf{f}_i(\mathbf{u}) = \mathbf{g}(\mathbf{b}) \Rightarrow \mathbf{u} \ll \mathbf{K}_i(\mathbf{b}) \tag{10}$$

- Thirdly, $(s_i, \mathbf{K}_i(\mathbf{b}))$ must precede $(t, \mathbf{b})$:

$$(s_i, \mathbf{K}_i(\mathbf{b})) \prec (t, \mathbf{b}); \tag{11}$$

- Lastly, $\mathbf{K}_i(\mathbf{b})$ must be a legitimate coordinate:

$$L_{s_i}\mathbf{K}_i(\mathbf{b}) + \mathbf{h}_{s_i} \geq 0. \tag{12}$$

In summary, if $\ll \max$ denote the lexicographic maximum of a set of integer lists:

$$\mathbf{K}_i(\mathbf{b}) = \ll \max \mathsf{Q}_i(\mathbf{b}) \tag{13}$$

where $\mathsf{Q}_i(\mathbf{b})$ is the set:

$$\mathsf{Q}_i(\mathbf{b}) = \{\mathbf{u} \quad | \quad \begin{array}{l} \mathbf{f}_i(\mathbf{u}) = \mathbf{g}(\mathbf{b}); \\ (s_i, \mathbf{u}) \prec (t, \mathbf{b}), \\ L_{s_i}\mathbf{u} + \mathbf{h}_{s_i} \geq 0\} \end{array} \tag{14}$$

with the convention that the lexical maximum of the empty set is undefined.

One should note the similarity of the above problem with the dependency computation. In fact, there is a dependency between s_i and t precisely when there exists a value of $\mathbf{b}$ such that $\mathsf{Q}_i(\mathbf{b})$ is not empty. Hence, the search for $\mathbf{K}_i$ may be limited to those instructions s_i such that there actually exists a dependency from s_i to t.

3.2.3 Evaluation techniques

In this paragraph, we will focus on one particular s_i (i.e. we will dispense with the index i). When the source program conforms to the restrictions of paragraph 2, all terms in formula (14) are linear equalities or inequalities. In fact since indexing functions are affine, the first term is a linear system whose dimension is that of array `A`. The last term is simply a set of linear inequalities. The second term may be computed with the help of (3). According to the definition of the lexicographic order, it is a disjunction of $N_{st} + 1$ terms. The term at depth p (according to [AK84] definition of depth) is:

$$\mathbf{u}[1..p] = \mathbf{b}[1..p] \wedge \mathbf{u}[p+1] < \mathbf{b}[p+1],$$

while the last one is:

$$\mathbf{u}[1..N_{st}] = \mathbf{b}[1..N_{st}].$$

This term must be omitted if T_{st} is false.

This means that the set $\mathsf{Q}(\mathbf{b})$ splits into $N_{st} + 1$ sets $\mathsf{Q}_p(\mathbf{b})$. There is a maximum for each non empty such set; we are interested in the latest one. However, no comparison is necessary since:

$$p < q \Rightarrow \mathsf{Q}_p(\mathbf{b}) \ll \mathsf{Q}_q(\mathbf{b}).$$

The solution lies in the non empty $\mathsf{Q}_p(\mathbf{b})$ with the highest index p. Now, since all constraints in the definition of $\mathsf{Q}_p(\mathbf{b})$:

$$\mathsf{Q}_p(\mathbf{b}) = \{\mathbf{u} \mid \mathbf{f}(\mathbf{u}) = \mathbf{g}(\mathbf{b}); \quad L_s\mathbf{u} + \mathbf{h}_s \geq 0, \quad \mathbf{u}[1..p] = \mathbf{b}[1..p], \mathbf{u}[p+1] < \mathbf{b}[p+1]\}, \tag{15}$$

are linear equalities and inequalities, $\mathsf{Q}_p(\mathbf{b})$ is the integer hull of a polyhedron. Finding its lexical maximum is a parametric integer problem, for which the author has devised an efficient algorithm in [Fea88b]. The parameters are the components of $\mathbf{b}$ and other integer variables (e.g. the variables which occurs in the array bounds). In the sequel, in the interest of legibility, we will not note explicitly these supplementary variables. Note also that the components of $\mathbf{b}$ are not arbitrary; they must satisfy various constraints, among which is:

$$L_t\mathbf{b} + \mathbf{h}_t \geq 0.$$

In [Fea88b] terminology, these inequalities form the context of the parametric integer problem.

To express the solution, we need the concept of a quasi-linear form. A quasi-linear form is constructed from the parameters and integer constants by the operations of addition, multiplication by an integer, and euclidean division by an integer. The name "quasi-linear" stems from the fact that is is possible to replace the integer quotients by extra parameters; the form is linear in the extended set of parameters, and the quotients may be defined by two supplementary linear constraints.

The solution is then expressed as a multistage conditional expression. The predicates are of the form $f(\mathbf{b}) \geq 0$, where f is quasi-linear. The values are quasi-linear forms or the "undefined" sign, ∞. The lexical maximum of $\mathsf{Q}_p(\mathbf{b})$ will be noted $\mathbf{k}_p(\mathbf{b})$. From the $\mathbf{k}_p$'s the construction of $\mathbf{K}$ is done by the following algorithm:

Algorithm (K)

1. let $p = N_{rs} - 1$ and $\mathbf{K} = \mathbf{k}_{N_{rs}}$;
2. if there are no occurences of ∞ in $\mathbf{K}$, stop;
3. otherwise, replace all occurences of ∞ by $\mathbf{k}_p$;
4. if $p = 0$ stop;
5. otherwise decrease p by 1 and go back to step (2).

$\mathbf{K}$ may sometime be simplified by detecting non-compatible predicates in the path from the root to a leaf. A set of quasi-linear predicates is non-compatible if the corresponding set of linear inequalities is not feasible. This is easily tested by the methods of [Fea88b]. If a non-compatible path is detected, simply delete the leaf and the last test.

We now put together all partial results to obtain the correct expression for $\mathtt{A}[\mathbf{g}(\mathbf{b})]$. We first build the following predicates:

$$\begin{aligned} P_i &= \bigwedge_{j>i} (s_j, \mathbf{K}_j(\mathbf{b})) \prec (s_i, \mathbf{K}_i(\mathbf{b})); \\ P_n &= \textbf{true} \ . \end{aligned} \tag{16}$$

The conjunction:

$$\neg P_1 \wedge \ldots \neg P_i$$

express the fact that s_i is the latest source for $\mathtt{A}[\mathbf{g}(\mathbf{b})]$. If instruction s_i has not been modified by the renaming process, let:

$$e_i = \mathtt{A}[\mathbf{f}_i(\mathbf{K}_i(\mathbf{b}))] = \mathtt{A}[\mathbf{g}(\mathbf{b})] \tag{17}$$

by (9). If instruction s_i has been subjected to Array Expansion, let:

$$e_i = \mathtt{A'}[\mathbf{K}_i(\mathbf{b})]. \tag{18}$$

Lastly, if s_i has been modified by Array Renaming, let:

$$e_i = \mathtt{A'}[\mathbf{g}(\mathbf{b})] \tag{19}$$

Replace $\mathtt{A}[\mathbf{g}(\mathbf{b})]$ by:

$$\begin{aligned} e \;=\; & \textbf{if } P_1 \textbf{ then } e_1 && (20) \\ & \textbf{else if } P_2 \textbf{ then } e_2 && \\ & \ldots && (21) \\ & \textbf{else } e_n. && (22) \end{aligned}$$

In building e, one should take into account as much simplifications as possible. Let us go back, for instance, to example {A} as rewritten into {B}.

There are two possible sources for $\mathtt{c}$ in the rhs of s, r and s itself. In the case of r, we must evaluate the function:

$$\mathbf{K}_r(i,j) = \ll \max \mathsf{Q}_r(i,j)$$

where:

$$\mathsf{Q}_r(i,j) = \{x | i + j = x; (r, x) \prec (s, i, j); 0 \leq x \leq n + m\}$$

in the context $0 \leq i \leq m, 0 \leq j \leq n$.

In this case, there is no need to use a complicated mathematical programming algorithm. $(r, x) \prec (s, i, j)$ is always true as we have already noted. The value

$x = i + j$ is the only solution of the first constraint, and it satisfies the second constraint as a consequence of the context. Hence the solution is

$$\mathbf{K}_r(i,j) = i + j,$$

always defined.

In the case of s:

$$\mathbf{K}_s(i,j) = \ll \max \mathsf{Q}_s(i,j)$$

$$\mathsf{Q}_s(i,j) = \{x, y | i + j = x + y; 0 \leq x \leq m; 0 \leq y \leq n; i > x \vee (i = x \wedge j > y)\}$$

The subset $Q_{s1}(i,j)$ is empty, since the system:

$$\begin{array}{rcl} i + j & = & x + y \\ i & = & x \\ j & < & y \end{array}$$

has no solution. Q_{s0} is the set:

$$\mathsf{Q}_{s0}(i,j) = \{y, z | i + j = y + z; 0 \leq y \leq m; 0 \leq z \leq n; y < i\}$$

in the context $0 \leq i \leq m, 0 \leq j \leq n$.

In order to apply [Fea88a] algorithm, we must first convert all constraints to the form $f \geq 0$, and transform the problem to a search for a lexicographic minimum. This is done by the following change of variables:

$$\begin{array}{rcl} 0 \leq y' & = & m - y \leq n + m \\ 0 \leq z' & = & n - z \leq n. \end{array}$$

The problem is cast in the form of a tableau:

$$\begin{array}{rcccccccl} y' = & y' & & & & & & & \geq 0, \\ z' = & & z' & & & & & & \geq 0, \\ a = & -y' & -z' & & -i & -j & +m & +n & \geq 0, \\ b = & y' & +z' & & +i & +j & -m & -n & \geq 0, \\ c = & -y' & & & & & +m & & \geq 0, \\ d = & & -z' & & & & & +n & \geq 0, \\ e = & y' & & -1 & +i & & -m & & \geq 0. \end{array}$$

When the context is taken into account, the algorithm is able to prove that the "parametric term" in row b (i.e. the part of row b which depends only on the parameters), is never positive. Hence b must becomes an independent variable in place of z'. The result is:

$$\begin{array}{rrrrrrrr}
y' = & y' & , & & & & & \\
z' = & -y' & +b & & -i & -j & +m & +n, \\
a = & & -b & , & & & & \\
b = & & b & , & & & & \\
c = & -y' & & & & & +m, & \\
d = & y' & -b & & +i & +j & -m, & \\
e = & y' & & -1 & +i & & m. &
\end{array}$$

Here again, the context shows that $-1+i-m<0$; y' must be eliminated in favour of e:

$$\begin{array}{rrrrrrrr}
y' = & e & & +1 & -i & & +m, & \\
z' = & -e & +b & -1 & & -j & & +n, \\
a = & & -b, & & & & & \\
b = & & b, & & & & & \\
c = & -e & & -1 & +i, & & & \\
d = & e & -b & +1 & & +j, & & \\
e = & e & . & & & & &
\end{array}$$

In this case, the context does not determine the sign of $n-j-1$. We must distinguish two cases. Similarly, $i-1$ may or may not be positive. If both linear forms are non-negative, all parametric terms are non negative integers, and the solution is found by zeroing the independent variables:

$$y' = m-i+1, z' = n-j-1,$$

or

$$\begin{aligned} y &= i-1, \\ z &= j+1. \end{aligned}$$

(In more complicated cases, the parametric terms may be fractional; one may have to introduce cuts to restore integrity, see [Fea88b] for details).

Suppose now that $i-1$ is negative. Inspection of the c row shows that the coefficients of the independent variables are null or negative. Hence bringing the row to a non-negative value is impossible, and the problem has no solution. After one more change of variable, a similar result is obtained in the case $n-j-1<0$. The final result is:

$$\mathbf{k}_0(i,j) = \textbf{if } (n-j-1>0) \wedge (i-1>0) \textbf{ then } \begin{pmatrix} i-1 \\ j+1 \end{pmatrix} \textbf{ else } \infty.$$

Since $\mathbf{k}_1(i,j)=\infty$, by algorithm $(\mathbf{K})$, $\mathbf{K}_s(i,j)=\mathbf{k}_0(i,j)$. There are only two selection predicates, P_r and $P_s = \textbf{true}$.

$$P_r = (r, [i+j]) \prec (s, [y, z])$$

where

$$y = \mathbf{if}\ (n-j-1>0) \wedge (i-1>0)\,\mathbf{then}\ i-1\,\mathbf{else}\ \infty,$$

and

$$z = \mathbf{if}\ (n-j-1>0) \wedge (i-1>0)\,\mathbf{then}\ j+1\,\mathbf{else}\ \infty.$$

We know that $(r, [x]) \prec (s, [y, z])$ is always true unless $[y, z]$ is undefined. Hence:

$$P_r = (n-j-1<0) \vee (i-1<0),$$

$$e_r = \mathsf{c}[i+j],$$

$$e_s = \mathsf{cc}[i-1, j+1].$$

We have simplified the expression of e_s, since it is used only in case $\mathbf{K}_s(i, j)$ is defined. Putting all this together, we obtain the final form of s:

```
cc[i,j] = (if (n-j-1 < 0) or (i-1 < 0)
             then c [i+j]
             else cc [i-1, j+1])
         + a [i]*b [j];
```

One may further replace `c[i+j]` by 0, but this is another type of program transformation. Note also that since this expression has no occurence of ∞, we have just proved that the original program is correct in so far as no element of `c` is used before being defined.

4 Scalar expansion revisited

In the case where the variable which carries the PP dependency is a scalar, the problem is much simpler. In the definition of $\mathsf{Q}_i(\mathbf{b})$, (see (14)), the first constraint is void, since a scalar has no indices. We are left with:

$$\mathsf{Q}_i(\mathbf{b}) = \{\mathbf{u} | (s_i, \mathbf{u}) \prec (t, \mathbf{b}); L_{s_i}\mathbf{u} + \mathbf{h}_{s_i} \geq 0\}, \tag{23}$$

in the context $L_t\mathbf{b} + \mathbf{h}_t \geq 0$.

There are $n = N_{s_i t}$ loops which surround both s_i and t and $N = N_{s_i s_i}$ loops which surround s_i alone.. Hence part of the constraint $L_{s_i}\mathbf{u} + \mathbf{h}_{s_i} \geq 0$ and part of the context are of the form:

$$lb_p \leq \mathbf{u}[p] \leq ub_p, 1 \leq p \leq n, \tag{24}$$

$$lb_p \leq \mathbf{b}[p] \leq ub_p, 1 \leq p \leq N. \tag{25}$$

We will suppose in the sequel that each loop is executed at least once, i.e. that:

$$lb_p \leq ub_p, 1 \leq p \leq n. \tag{26}$$

It is left to the reader to add the necessary conditional prefixes in case these conditions cannot be proved from the program text.

There are two cases: either $T_{s_i t}$ is true or not. In the first case, the last term in the sequencing predicate is:

$$\mathbf{u}[1..N_{s_i t}] = \mathbf{b}[1..n], \tag{27}$$

and we claim that the solution is:

$$\mathbf{K}_{s_i}(\mathbf{b}) = [\mathbf{b}[1..n], ub_{n+1}, \ldots, ub_N]. \tag{28}$$

This obviously verify (27); (24) is verified as a consequence of (25). Furthermore, there are no other possible values for the n first components of $\mathbf{K}_{s_i}$, and the next components are as large as possible.

For readability sake, (28) has been written as if the upper bounds were constants; it is left to the reader to see that if the bounds are affine forms, they may be evaluated from left to right in term of $\mathbf{b}[1..n]$.

Suppose now that $T_{s_i t}$ is false; the sequencing predicate at depth p is:

$$\mathbf{u}[1..p] = \mathbf{b}[1..p]; \mathbf{u}[p+1] < \mathbf{b}[p+1]. \tag{29}$$

A possible solution is:

$$\mathbf{K}_p(\mathbf{b}) = [\mathbf{b}[1..p], \mathbf{b}[p+1] - 1, ub_{p+2}, \ldots, ub_N]. \tag{30}$$

This is feasible if and only if $\mathbf{b}[p+1] - 1 \geq lb_{p+1}$, which is not guaranteed by the context. Execution of algorith ($\mathbf{K}$) gives the final solution:

$$\begin{aligned}
\mathbf{K}_{s_i}(\mathbf{b}) \quad = \quad & \textbf{if } \mathbf{b}[n] \geq lb_n & (31)\\
& \textbf{then } [\mathbf{b}[1..n-1], \mathbf{b}[n] - 1, ub_{n+1}, \ldots] \\
& \textbf{else if } \mathbf{b}[n-1] \geq lb_{n-1} \\
& \textbf{then } [\mathbf{b}[1..n-2], \mathbf{b}[n-1] - 1, ub_n, \ldots] \\
& \textbf{else } \ldots \\
& \textbf{else } \infty. & (32)
\end{aligned}$$

From (28) and (31) a complete solution may be developped. In the special case where we are interested only in one loop and one scalar, it is easy to see that application of (28) and (31) is equivalent to the rules of 1.3.1. But our methods are much more powerfull. Take for example the case of the matrix product:

```
for i := 1 to n do
    for j := 1 to n do
    begin
{r}    v := 0;
       for k := 1 to n do
{s}        v := v + a[i,k] * b[k,j];       {C}
       c[i,j] := v;
    end;
```

Suppose we decide to promote `v` to a two dimensional array `vv` with indices `i` and `j` both in r and s. A straightforward application of (28) and (31), plus some simplifications, gives the following result:

```
for i := 1 to n do
    for j := 1 to n do
    begin
{r}     vv[i,j] := 0;             {D}
        for k := 1 to n do
{s}         vv[i,j] := vv[i,j] + a[i,k] * b[k,j];
        c[i,j] := vv[i,j];
    end;
```

It would be possible to extend the rules in 1.3.1 to cover such cases, but the result would be very ungainly; application of our technique is probably much simpler.

5 Conclusion

We have given a general technique which allows an automatic restructuring compiler to expand or rename an array definition in order to avoid output dependencies. This technique involves solving small parametric integer problems. The algorithm of [Fea88b] has been implemented and found to work reliably on examples such as programs {A} and {C}. The method will work under any expansion strategy: one simply has to modify accordingly rules (17) to (19).

The overall complexity of the method depends on several size parameters. On the one hand, the amount of work is proportional to the number of references to be rewritten, i.e. to the number of data dependencies originating in the expanded instruction, t.

On the other hand, at each reference s one must solve N_{ss} Parametric Integer Problems whose size depends on:

- the number of unknowns, n, which is N_{ss};

- the number of parameters, p, which is N_{tt} plus the number of auxilliary parameters (loop limits, etc.);
- the number of constraints, m: there is one equality per dimension of the expanded array, plus $2N_{ss}$ loop limit constraints, plus $d+1$ sequencing constraints where d is the depth.

In real-life programs, all these values are small integers: for instance, arrays seldom have more than 3 indices. The algorithm proceeds mainly by pivoting steps in the manner of the Gauss-Jordan algorithm, each step having an operation count proportional to $(n+p)m$. In theory, the step count may be quite large: the simplex is not polynomial. However, it is well known that for most data, the simplex is polynomial with a pivoting count proportional to the number of unknowns. Preliminary experience sustains the conjecture that the Parametric Integer algorithm shares this property.

There are several points which deserve further study. Before real size application is contemplated, systematic techniques for simplifying formulas such as (20) should be devised. Similarly, techniques for avoiding useless computations (e.g., distinguishing between live and dead definitions), would be very useful. It would also be very interesting to lift some of the restrictions we imposed in 2.1 (e.g., to allow conditionals instructions).

There are many degrees of freedom in the choice of rewriting or expansion beyond the simple rules of 3.1. How does one judge a priori the interest of such a transformation? Are there situations in which the data flow reconstruction is simplified? In the case of $\{C\}$, this is probably because all use of the offending variable where rewritten in the same way. Is this always possible?

We believe the techniques we have introduced in this paper to be susceptible of many applications beyond array expansion. We have noted that they give a method for testing that no array element is used before being defined. Similarly, the method may be used to build assertions on arrays, in the manner of [Jou87]. It may even be possible in some cases to solve recurrence equations on array, as for instance to transform:

```
x[0] := 0;
for i := 1 to n do
    x[i] := x[i-1];
```

into

```
for i := 0 to n do
    x[i] := 0;
```

with the attendant increase in parallelism. Finally, let us note that our method would give very interesting information to an optimizing compiler.

References

[AK84] J.R. Allen and Ken Kennedy. Automatic loop interchange. In *Proc. of the 1984 ACM SIGPLAN Compiler Conference*, pages 233–246, June 1984.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass, 1986.

[Bak77] Brenda S. Baker. An algorithm for structuring programs. *Journal of the ACM*, 24:98–120, 1977.

[CH78] Patrick Cousot and N. Halbwach. Automatic discovery of linear restraints among variables of a program. In *ACM POPL*, 1978.

[Dij68] Edger W. Dijkstra. Goto statements considered harmfull. *Communications of the ACM*, 11:147–148, 1968.

[Fea88a] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, 1988.

[Fea88b] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Jou87] Pierre Jouvelot. Semantic parallelization, a practical exercise in abstract interpretation. In *ACM POPL*, 1987.

[Kuc78] David J. Kuck. *The Structure of Computers and Computations.* J. Wiley and sons, New York, 1978.

[PW86] D. A. Padua and Michael J. Wolfe. Advanced compiler optimization for supercomputers. *CACM*, 29:1184–1201, December 1986.

[Wol78] Michael J. Wolfe. *Techniques for improving the inherent parallelism in programs.* Master's thesis, Univ. of Illinois at Urbana-Champlain, 1978.