Dataflow Analysis of Array and Scalar References

Paul Feautrier*

September 1991

Abstract

Given a program written in a simple imperative language (assignment statements, `for` loops, affine indices and loop limits), this paper presents an algorithm for analyzing the patterns along which values flow as the execution proceeds. For each array or scalar reference, the result is the name and iteration vector of the source statement as a function of the iteration vector of the referencing statement. The paper discusses several applications of the method: conversion of a program to a set of recurrence equations, array and scalar expansion, program verification and parallel program construction.

Keywords dataflow analysis, semantics analysis, array expansion.

1 Introduction

It is a well known fact that scientific programs spend most of their running time in executing loops operating on arrays. Hence if a restructuring or optimizing compiler is to do a good job, it must be able to do a thorough analysis of the addressing patterns in such loops. If taken in full generality, the analysis problem is intractable. In this paper, we consider a class of programs for which this analysis is possible : programs with so-called static

*Laboratoire MASI, Université P. et M. Curie, 75252 PARIS CEDEX 05 FRANCE, e-mail: `feautrier@masi.ibp.fr`

control and affine indices. There are reasons to believe that a large proportion of all numerical programs belongs to this class, and that many more may be converted to it by appropriate preprocessing. The analysis of addressing patterns in this class may be reduced to the solution of parametric systems of linear inequalities in integers, for which the author has devised an efficient algorithm [14].

The central problem to be solved here is the following: given an array cell, which of several statements is the source of the value contained therein at a given instant in the execution of a program. Most of the time, the statement will be embedded in a loop nest. Hence, we will require not only the name of the source statement, but also the values of the loop counters at the time the value of interest was generated. This information may be packaged as a *source function*, as the source will depend on the iteration vector of the destination. We will give here a solution for programs with `for` loops as the only control statement. As a particular case, our method gives a general solution to the problem of the source of scalars, which may be seen as degenerate arrays with no indices. A knowledge of the source function allows one to solve many problems which include automatic translation to single assignment form, array and scalar expansion, dead code elimination, and various questions connected to the construction of programs for vector and parallel processors.

1.1 Outline

Section 2 describes the simple programming language we will use for giving examples and the necessary restrictions on its indexing functions and loop limits. We will also introduce the sequencing predicate as a compact notation for deciding which of two statement instances is executed first. Section 3 is the central part of the paper; here we give a detailed account of the dataflow computation. Section 4 outlines in varying detail several applications of the technique. Section 5 lists some previous results which may be seen as particular cases of the methods we have introduced in section 3.

In the conclusion, we give some empirical evidence on the complexity of the algorithm and point to several possible extensions. The parametric integer algorithm, which is a basic component of the present method, is summarized in the appendix. For a more detailed presentation and proofs the reader is referred to the above quoted paper [14].

1.2 Notations

Bold letters will denote vectors or vector valued functions; $|\mathbf{a}|$ is the dimension of vector $\mathbf{a}$. $\mathbf{a}[i..j]$ is the subvector of $\mathbf{a}$ built from components i to j. $\mathbf{a}[i]$ is a shorthand for $\mathbf{a}[i..i]$. Familiar operators and predicates like $+$ and $\geq$ will be tacitly extended to vectors. The sign $\ll$ will denote lexical ordering of vectors. Large letters will usually denote sets; N will be the set of non-negative integers. If A is a matrix, A_{ij} will be its generic element, $A_{i\bullet}$ its generic row and $A_{\bullet j}$ its generic column.

2 The Program Model

In this section, we will first describe the syntax of the source language. We will then discuss the restrictions we superimpose on this syntax. In the following development, we will distinguish between *statements*, which are syntactic parts of the program text, and *operations*, which are actions inducing modifications of the computer store. Most often, a statement will be executed several times, giving rise to many distinct operations. We will introduce the sequencing predicate as a means of specifying the execution order of operations.

2.1 The Source Language

The source language may be seen either as a static PASCAL or as a rationalized FORTRAN. In fact, our work is not about any particular language, but about the *static subset* of most programming languages, i.e. about what happens when all memory allocation has been taken care of. Data types will be restricted to integers, reals, and n-dimensional arrays of integers and reals. The only simple statements we will consider will be scalar and array assignments. The only control constructs will be the sequence and the `for` loop. We will extend the language in order to allow conditional expressions (à la Algol 60), which are necessary for the expression of index calculations (see e.g. section 3.3). The syntax will be:

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

Note the absence of `goto`'s, of conditional statements, of `while` loops and of procedures.

2.2 Restrictions

To be able to analyze array accesses inside loops, one must have some knowledge of the iteration count of these loops. The simplest case is when limits are known numerical values. This, however, is much too restrictive, since many programs use variable limits (matrix and vector dimensions, discretization size, etc.) and even non rectangular loop nests: consider for instance the prevalence in numerical analysis of triangularization algorithms (like those of Gauss or Cholesky). To extend the class of tractable programs, we will introduce the notion of static control.

To recognize a static control program, one must first identify its structure parameters: a set of integer variables which are defined only once in the program, and whose value depends only on the outside world (through an input statement) or on other already defined structure parameters. A program has static control if all its loops are `for` loops whose limits depend only on structure parameters, numerical constants and outer loops iteration counters. The analysis technique which is presented here is applicable only if all loops have increment 1, and if all limits are affine functions. For similar reasons, all indices will be restricted to affine functions of the loop counters and the structure parameters.

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 their indices are equal. This restriction is not too severe if we note, first, that it is good programming practice to debug a program before submitting it to an optimizing or restructuring compiler, and also that the methods of this paper may be used as a highly efficient array access checker [24].

This hypothesis will allow us to ignore array declarations. As a consequence, our technique will be equally applicable to languages which enforce constant array bounds – Fortran, Pascal, C, ... – and to those which do not.

2.3 The Sequencing Predicate

Values in array elements are produced by execution of statements. Hence we need a notation to pin-point a specific execution of a statement, or *operation*. Our first need is an unambiguous designation of a statement in a program. Neither the text of the statement nor its position in the program syntax tree will serve, since there may be several statements with the same text, and since the program may be modified by a restructuring compiler. Hence we will use a set of arbitrary statement 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 statement descriptions. In the balance of this paper, we will mostly be interested in simple statements. However, some discussions will be clearer if all statements, compound or simple, are named.

In our source language, the only repetitive construct is the `for` loop. Hence, an operation is uniquely defined by the name of the statement and the values of the surrounding loop counters (the *iteration vector* [17]). A pair such as $(r, \mathbf{a})$ whose components are a statement name and an integer vector will be called an (operation) coordinate. To denote a statement instance, a coordinate must satisfy two conditions:

- the dimension of $\mathbf{a}$ must be equal to the number of loops surrounding r;
- all components of $\mathbf{a}$ must be within the corresponding loop limits.

With each loop t we may associate a pair of inequalities:

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

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

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

(1) may be summarized in matrix form as:

$$\mathbf{e}_r(\mathbf{a}) \geq 0. \tag{2}$$

where $\mathbf{e}_r$ is an affine vector-valued function. Formula (2) will be called the existence predicate of r. Notice that we do not suppose that $lb_t \leq ub_t$. In accordance with the Pascal convention (and with the "modern" interpretation of Fortran `DO` loops), a loop whose limits violate this inequality will not be executed at all.

Consider for example the program sketch in figure 1. Figure 2 describes its iteration domain. The existence predicate of statement s_2 may be written as:

$$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \end{pmatrix} \geq 0.$$

One should not infer from figure 2 that all statements have iteration domains which lies in the same euclidean space. As a counter-example, consider the program of figure 3. As shown in figure 4, s_1 has a one-dimensional iteration domain, while s_2 has a two-dimensional one.

Finally, one should not confuse the iteration domain, which is spanned by loop counters, and the data space, which is spanned by array indices. In many cases, those two spaces are identical (or rather, isomorphic) as in:

```
for i := 1 to n do
   for j := 1 to n do
      x[i,j] := 0.;
```

but this is not always true. In the case of the program in figure 5, the iteration domain is two-dimensional while the data space is one-dimensional. Conversely, in:

```
for i := 1 to n do t[i,i] := 1.;
```

the data space is a one-dimensional subspace embedded in a two-dimensional space.

```
for i:=1 to n
begin for j := 1 to i-1 do S1;
      for j := i+1 to n do S2;
end;
```

Figure 1: A sample program

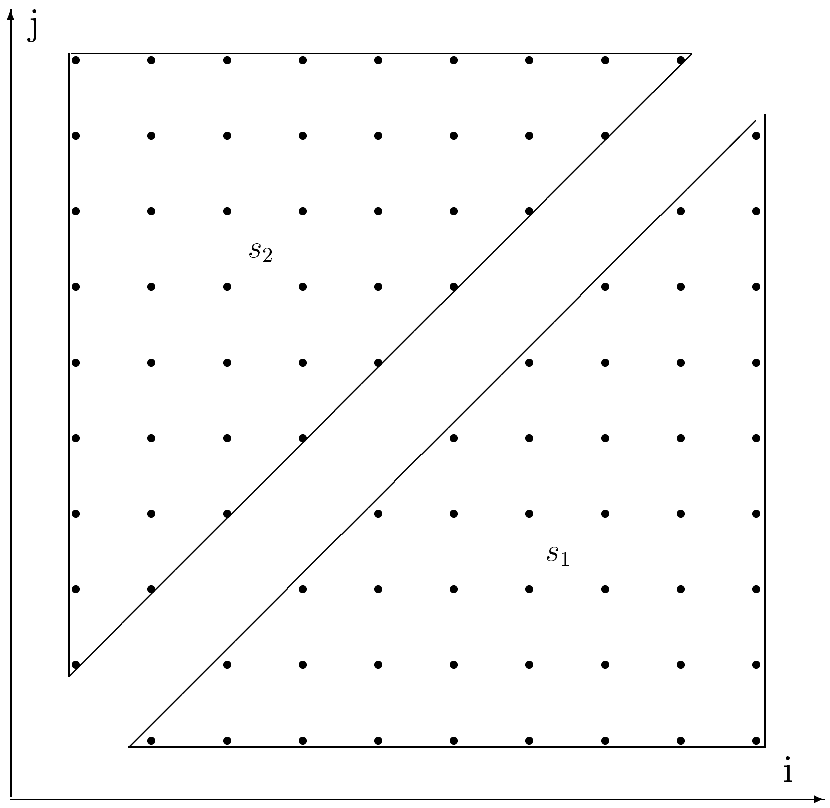

Figure 2: The iteration domain of program 1

```
for i := 1 to n do
begin
  x[i]:=0.;                    {S1}
  for j := 1 to i do
     x[i] := x[i] + u[i,j] * y[j] {S2}
end;
```

Figure 3: An imperfectly nested program

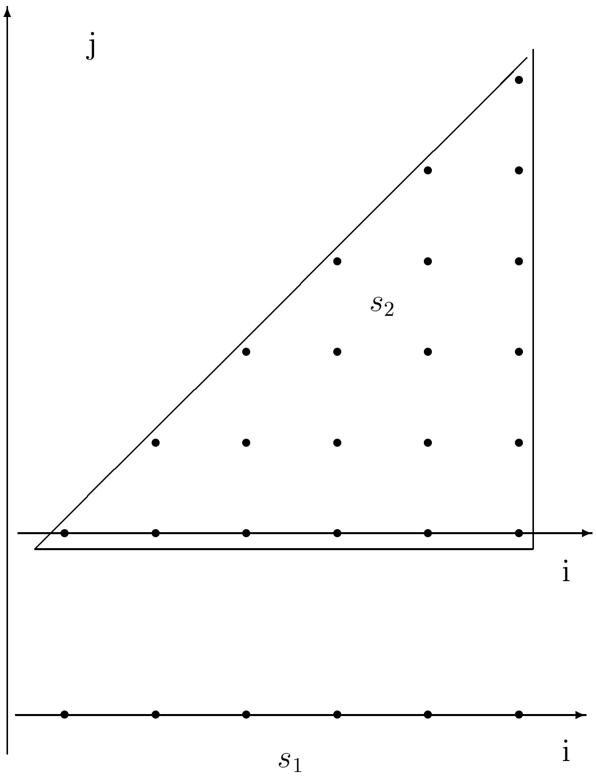

Figure 4: The iteration domain of program 3

```
for k := 0 to 2*n do
   c[k] := 0.;                         {S1}
for i := 0 to n do
   for j := 0 to n do
      c[i+j] := c[i+j] + a[i]*b[j];   {S2}
```

Figure 5: The product of two polynomials

The preceding discussion leads to a spatial description of loops. Such a point of view goes back to the work of Kuck; see also Padua and Wolfe's review article [20]. Usually, loops are explained from a temporal point of view: iteration i is executed just before iteration $i+1$. We must seek a way to reconcile those two aspects. This may be done by defining a *sequencing predicate* on the iteration domains. The sequencing predicate is a strict total order on the set of operation coordinates; it is written:

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

and expresses the fact that $(r, \mathbf{a})$ is executed before $(s, \mathbf{b})$. The sequencing predicate depends only on the source program text. Our present aim is to give a simple expression for it.

Suppose first that r and s are statements in the outermost statement list of the program. $\mathbf{a}$ and $\mathbf{b}$ necessarily are the zero dimensional vector $[]$. $(r, []) \prec (s, [])$ iff r precedes s in the program text. Let T_{rs} be a boolean which is true iff r textually precedes s; in this case:

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

Note that T_{rr} is false and that if $r \neq s$ then $T_{rs} \equiv \neg T_{sr}$.

Next, suppose that r and s are the same statement. In this case, according to the familiar semantics of `for` loops, $(r, \mathbf{a}) \prec (r, \mathbf{b})$ iff $\mathbf{a}$ is lexicographically smaller than $\mathbf{b}$.

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 statements r' and s' such that r is r' or is textually 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 $\mathbf{b}[1..N_{rs}]$. Otherwise, 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 $T_{r's'} = T_{rs}$. Putting all this together:

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

Knowledge of N_{rs} (a matrix of integers) and T_{rs} (a matrix of booleans) is all that is needed to sequence all operations in a program.

When lexicographic order is replaced by its definition, the sequencing predicate becomes a disjunction of $N_{rs} + 1$ affine predicates which will be written as $\prec_p$:

$$(r, \mathbf{a}) \prec_p (s, \mathbf{b}) \equiv (\mathbf{a}[1..p] = \mathbf{b}[1..p] \wedge \mathbf{a}[p+1] < \mathbf{b}[p+1]), \quad 0 \leq p < N_{rs}. \quad (4)$$

The version for $p = N_{rs}$ is :

$$(r, \mathbf{a}) \prec_p (s, \mathbf{b}) \equiv \mathbf{a}[1..N_{rs}] = \mathbf{b}[1..N_{rs}] \wedge T_{rs}. \quad (5)$$

One may notice that operations which stand in the relation $\prec_p$ to each other have exactly p identical coordinates in their iteration vectors. In Allen and Kennedy's paper[3], if two such operations give rise to a dependence, one says that this dependence is at depth $p+1$, while if $p = N_{rs}$, the depth is said to be infinite. With a slight displacement of the origin, we will say that $\prec_p$ is the sequencing predicate at depth p, depths ranging from 0 to N_{rs}.

Consider again the program of figure 3. The sequencing between s_1 and s_2 is given by $N_{s_1 s_2} = 1$ and $T_{s_1 s_2} = \mathbf{true}$. Hence:

$$(s_1, i) \prec (s_2, i', j') \equiv i < i' \vee i = i'. \quad (6)$$

Similarly, the sequencing between two instances of s_2 is given by:

$$(s_2, i, j) \prec (s_2, i', j') \equiv i < i' \vee (i = i' \wedge j < j'), \quad (7)$$

since $T_{s_2 s_2}$ is false.

These results may be summarized by figure 6. In this diagram, we have only represented essential edges of the $\prec$ relation. All other edges may be recovered by using the transitivity of $\prec$.

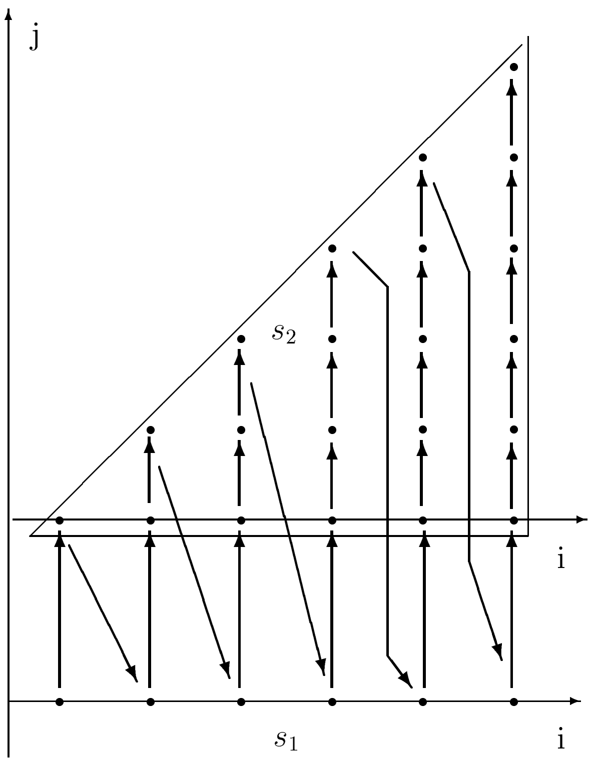

Figure 6: The sequencing predicate of loop 3

3 Data Flow Analysis

3.1 Some Notation

Suppose that we are given a program conforming to the restrictions of section 2.2. Let t be a statement in which an array $\mathbf{M}$ is used. Let $\mathbf{b}$ be the iteration vector of t; the indices of $\mathbf{M}$ are affine functions of $\mathbf{b}$. In vector form, the reference to $\mathbf{M}$ may be written $\mathbf{M}[\mathbf{g}(\mathbf{b})]$.

> Consider for instance the reference to `v[i,k]` in:
>
> ```
> for i := 1 to n do
>    for j := 1 to i-1 do
>       for k := i+1 to n do
>          v[j,k] := v[j,k]-v[i,k]*v[j,i]/v[i,i];
> ```
>
> The surrounding loop counters are i, j and k. The indexing function, $\mathbf{g}$, is given by:
>
> $$\mathbf{g}(i,j,k) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix}.$$
>
> The indexing function is exactly what is needed to connect the iteration domain to the data space.

We are interested in finding the source of the value of $\mathbf{M}[\mathbf{g}(\mathbf{b})]$. Let $s_1, s_2, \ldots, s_n$ be the statements which produce a value for $\mathbf{M}$, and let $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_n$ be their iteration vectors. s_i is of the form:

$$\mathbf{M}[\mathbf{f}_i(\mathbf{a}_i)] := \cdots.$$

The source is a coordinate, or rather a function of $\mathbf{b}$ which gives a coordinate when evaluated, which will be called the source function of $\mathbf{M}[\mathbf{g}(\mathbf{b})]$.

3.2 Formal Solution

If the source of $\mathbf{M}[\mathbf{g}(\mathbf{b})]$ is an instance of s_i, there is a unique $\mathbf{a}_i$ such that this instance is $(s_i, \mathbf{a}_i)$. This $\mathbf{a}_i$ is a function of $\mathbf{b}$, which will be called $\mathbf{K}_{s_i t}$. The real source is the latest operation $(s_i, \mathbf{K}_{s_i t}(\mathbf{b}))$:

$$\forall j \neq i, (s_j, \mathbf{K}_{s_j t}(\mathbf{b})) \prec (s_i, \mathbf{K}_{s_i t}(\mathbf{b}))$$

The correct value of i may depend on $\mathbf{b}$. In particular, $\mathbf{K}_{s_i t}(\mathbf{b})$ may be undefined for some values of $\mathbf{b}$. We will postulate that an undefined operation (written as $\perp$) comes earlier than any other operation:

$$\forall t, \mathbf{b} : \perp \prec (t, \mathbf{b}).$$

The conditions on $\mathbf{K}_{s_i t}(\mathbf{b})$ are:

- Firstly, $(s_i, \mathbf{K}_{s_i t}(\mathbf{b}))$ must produce a value for $\mathbf{M}[\mathbf{g}(\mathbf{b})]$:

$$\mathbf{f}_i(\mathbf{K}_{s_i t}(\mathbf{b})) = \mathbf{g}(\mathbf{b})$$

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

$$(s_i, \mathbf{K}_{s_i t}(\mathbf{b})) \prec (t, \mathbf{b});$$

- Thirdly, $\mathbf{K}_{s_i t}(\mathbf{b})$ must be a legal coordinate:

$$\mathbf{e}_{s_i}(\mathbf{K}_{s_i t}(\mathbf{b})) \geq 0.$$

- Lastly, $(s_i, \mathbf{K}_{s_i t}(\mathbf{b}))$ must be the latest coordinate which satisfies all conditions above:

$$\mathbf{f}_i(\mathbf{u}) = \mathbf{g}(\mathbf{b}) \wedge (s_i, \mathbf{u}) \prec (t, \mathbf{b}) \wedge \mathbf{e}_{s_i}(\mathbf{u}) \geq 0 \Rightarrow \mathbf{u} \underline{\ll} \mathbf{K}_{s_i t}(\mathbf{b});$$

In summary, letting $\max_{\ll}$ denote the lexicographic maximum of a set of integer vectors:

$$\mathbf{K}_{s_i t}(\mathbf{b}) = \max_{\ll} \mathsf{Q}_{s_i t}(\mathbf{b}) \tag{8}$$

where $\mathsf{Q}_{s_i t}(\mathbf{b})$ is the set:

$$\mathsf{Q}_{s_it}(\mathbf{b}) = \{\mathbf{u} | \mathbf{f}_i(\mathbf{u}) = \mathbf{g}(\mathbf{b}), (s_i, \mathbf{u}) \prec (t, \mathbf{b}), \mathbf{e}_{s_i}(\mathbf{u}) \geq 0\} \tag{9}$$

with the convention that the lexical maximum of the empty set is $\perp$.

Now, since $\prec$ is a disjunction of $N_{s_it} + 1$ linear predicates, Q_{s_it} is the union of $N_{s_it} + 1$ disjoint polyhedra, indexed by $p, 0 \leq p \leq N_{s_it}$:

$$\mathsf{Q}^p_{s_it}(\mathbf{b}) = \{\mathbf{u} | \mathbf{f}_i(\mathbf{u}) = \mathbf{g}(\mathbf{b}), (s_i, \mathbf{u}) \prec_p (t, \mathbf{b}), \mathbf{e}_{s_i}(\mathbf{u}) \geq 0\}, \tag{10}$$

$$\mathbf{K}^p_{s_it}(\mathbf{b}) = \max_{\ll} \mathsf{Q}^p_{s_it}(\mathbf{b}). \tag{11}$$

Finally, if $\max_{\prec}$ is the maximum according to the sequencing predicate, then the source is given by:

$$\mathbf{S} = \max_{\prec}\{(s_i, \mathbf{K}^p_{s_it}(\mathbf{b})) | i = 1, \ldots, n, p = 0, \ldots, N_{s_it}\}. \tag{12}$$

To avoid multiple indices, we will renumber all possible candidates at all depths with a new index j. L will stand for the cardinality of the set of possible sources. (12) will be rewritten as :

$$\mathbf{S} = \max_{\prec}\{(s_j, \mathbf{K}_j(\mathbf{b})) | j = 1, L\}. \tag{13}$$

Let us go back to the example in Figure 5. Consider the problem of finding the source of `c[i+j]` in statement s_2. There are two candidates, s_1 and s_2 itself, and as a consequence, two functions $\mathbf{K}_{s_1s_2}$ and $\mathbf{K}_{s_2s_2}$.

Consider for instance the set $\mathsf{Q}_{s_2s_2}(i,j)$. Its elements are two dimensional integer vectors (i', j') which satisfy the following constraints:

- the index equations, $i' + j' = i + j$;
- the sequencing constraint $i' < i \vee (i' = i \wedge j' < j)$. One sees that the second term in the disjunction is incompatible with the index equation.
- the limit constraints $0 \leq i' \leq n, 0 \leq j' \leq n$.

Examination of figure 7 shows that $\mathsf{Q}_{s_2s_2}(i,j)$ is empty if $i = 0$ or $j = n$. If not empty, its lexical maximum is the vector $(i-1, j+1)$. This implies that to represent $\mathbf{K}_{s_2s_2}$, we will need a conditional:

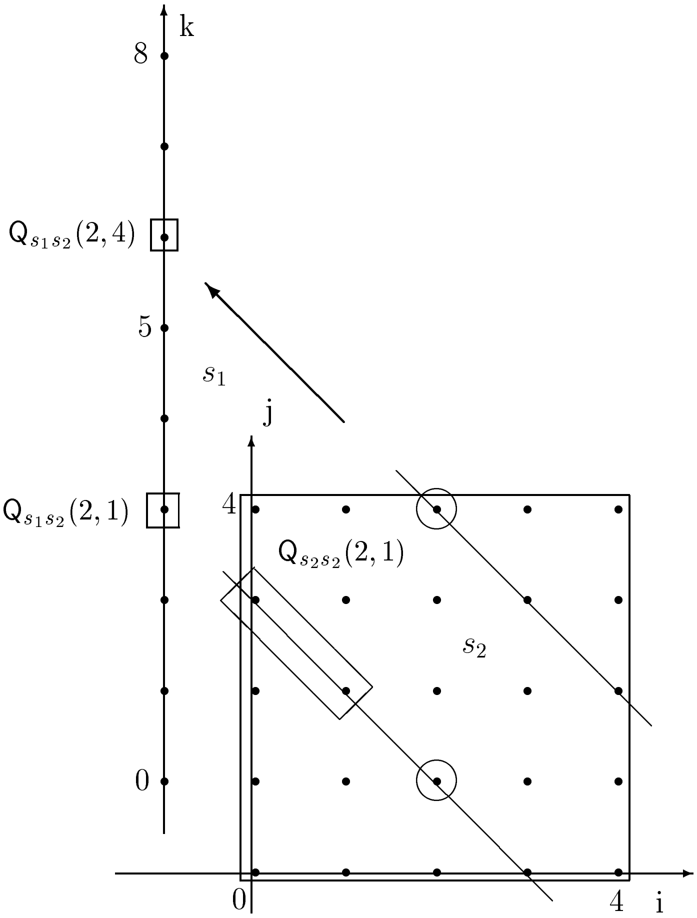

Figure 7: Computing the source function for the program of Figure 5

The problem is finding the source of `c[3]` at iteration $(2, 1)$ and of `c[6]` at iteration $(2, 4)$ (circled points).
Square boxes enclose the corresponding Q sets.

$$\mathbf{K}_{s_2 s_2}(i,j) = \mathbf{if}\ (i \geq 1 \wedge j < n)\ \mathbf{then}\ (i+1, j-1)\ \mathbf{else}\ \perp. \quad (14)$$

The case of the other candidate is simpler; we always have:

$$\mathbf{K}_{s_1 s_2}(i,j) = (i+j).$$

Now, it should be clear from an examination of program in figure 5 (or from the fact that $N_{s_1 s_2} = 0$ and that $T_{s_1 s_2}$ is true), that all operations (s_1, k') precede all operations (s_2, i', j'). It follows that the source is given by $\mathbf{K}_{s_2 s_2}(i,j)$ *provided this quantity is defined*. Hence, the final result is:

$$\mathbf{S}(i,j) = \mathbf{if}\ (i \geq 1 \wedge j < n)\ \mathbf{then}\ (s_2, i+1, j-1)\ \mathbf{else}\ (s_1, i+j). \quad (15)$$

To obtain this result, we have relied a lot on figure 7 and geometrical intuition. Now this works fine on one- and two-dimensional problems, but is quite difficult and error prone in three dimensions, and is impossible beyond. Furthermore, a computer has no geometrical intuition at all. Our aim now will be to solve the above problem in a general, systematic fashion and to implement the corresponding algorithm.

3.3 Evaluation Techniques

3.3.1 Direct Dependences

In this section, we will focus first on one particular candidate $(s_j, \mathbf{K}_j(\mathbf{b}))$ at a given depth p. When the original program conforms to the restrictions of section 2.2, all terms in formula (10) are linear equalities or inequalities. In fact since indexing functions are affine, the first term is a linear system whose dimension is the rank of array $\mathbf{M}$. The last term is simply a set of linear inequalities. The second term is given by (4) or (5). If the depth p is less than $N_{s_i t}$, then it is the conjunction of p equalities and one inequality. For $p = N_{s_i t}$, it is made of equalities only and does not exist if $T_{s_i t}$ is false.

As a consequence, $\mathsf{Q}_j(\mathbf{b})$ is the set of integer vectors which lie inside a polyhedron. Finding its lexical maximum is a Parametric Integer Program (a PIP)[14]. A short description of an algorithm for solving PIP problems is given in the appendix. The parameters are the components of $\mathbf{b}$ and the structure parameters. Note that the components of $\mathbf{b}$ are not arbitrary; they must satisfy various constraints, among which is:

$$\mathbf{e}_t(\mathbf{b}) \geq 0,$$

to which may be added any available information on the structure parameters. These inequalities form the *context* of the parametric integer problem.

To express the solution, we need the concept of a quasi-affine form. Such a form is constructed from the parameters and integer constants by the operations of addition, multiplication by an integer, and division by an integer. 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-affine. The leaves are vector of quasi-affine forms or the "undefined" sign, $\perp$. Such an expression will be called a quasi-affine selection tree (quast for brevity).

The above definition may be summarized by the following grammar:

$$\begin{array}{rl}
\text{form} ::= & \text{integer} \\
| & \text{parameter} \\
| & \text{integer} * \text{form} \\
| & \text{form} \div \text{integer} \\
| & \text{form} + \text{form} \\
\text{vector} ::= & (\text{form}[, \text{form}] \ldots) \\
\text{quast} ::= & \perp \\
| & \text{vector} \\
| & \textbf{if } \text{form} \geq 0 \textbf{ then } \text{quast} \textbf{ else } \text{quast}
\end{array}$$

The result of this analysis is the direct dependence between the definition by s_j and the use in t. Direct dependences were first defined by Brandes [10]. The presence of a $\perp$ sign in a direct dependence indicates that, for some values of the loop counters, the reference in t is not defined by statement s_j.

Formula (14) is a quast in the above sense (notice that integer division is not used here). Integer division appears when analyzing programs which access arrays with strides greater than one, as in:

```
s := 0.;
```

```
for i := 1 to n do
  x[2*i-1] := 1.;        {S1}
for k := 1 to 2*n-1 do
  s := s + x[k];         {S2}
```

The direct dependence from `x[2*i-1]` in s_1 to `x[k]` in s_2 is given by the following quast:

$$\mathbf{q} = \mathbf{if}\ 2((k+1) \div 2) - (k+1) \geq 0\ \mathbf{then}\ (k+1) \div 2\ \mathbf{else}\ \perp.$$

This formula expresses the fact that `x[k]` is not defined when k is even.

3.3.2 Combining the direct dependences

Consider now the problem of evaluating (13). This will be done in a sequential manner, by introducing:

$$\begin{aligned} \mathbf{S}_n &= \max_{\prec}\{(s_j, \mathbf{K}_j(\mathbf{b})) | j = 1, \ldots, n\}, \\ \mathbf{S}_0 &= \perp. \end{aligned}$$

Obviously, $\mathbf{S} = \mathbf{S}_L$ and we have the recurrence:

$$\mathbf{S}_n = \max_{\prec}\{\mathbf{S}_{n-1}, (s_n, \mathbf{K}_n(\mathbf{b}))\} \tag{16}$$

We are thus led to the evaluation of:

$$\mathbf{S} = \max_{\prec}\{\mathbf{T}, (s_n, \mathbf{K}_n(\mathbf{b}))\}, \tag{17}$$

where $\mathbf{T}$ is an arbitrary quast. There are three cases, according to the form of $\mathbf{T}$:

- $\mathbf{T} = \perp$; in this case:
$$\mathbf{S} = (s_n, \mathbf{K}_n(\mathbf{b})). \tag{18}$$
- $\mathbf{T} = \mathbf{if}\ \pi(\mathbf{b})\ \mathbf{then}\ \mathbf{T}_1\ \mathbf{else}\ \mathbf{T}_2$; in this case:
$$\begin{aligned} \mathbf{S} \ = \ & \mathbf{if}\ \pi(\mathbf{b}) \\ & \mathbf{then}\ \max_{\prec}\{\mathbf{T}_1, (s_n, \mathbf{K}_n(\mathbf{b}))\} \\ & \mathbf{else}\ \max_{\prec}\{\mathbf{T}_2, (s_n, \mathbf{K}_n(\mathbf{b}))\}. \end{aligned} \tag{19}$$

- $\mathbf{T} = (r, \mathbf{l}(\mathbf{b}))$ where r is a statement name and $\mathbf{l}$ is a quasi-affine form; then:

$$\mathbf{S} = \textbf{if } (r, \mathbf{l}(\mathbf{b})) \prec (s_n, \mathbf{K}_n(\mathbf{b}))\textbf{ then } (s_n, \mathbf{K}_n(\mathbf{b}))\textbf{ else } (r, \mathbf{l}(\mathbf{b})). \quad (20)$$

  In this formula, the sequencing predicate is to be expanded with the help of (3).

These rules (and their symmetric counterparts, as the max operator is commutative), are the basic tools for computing source functions. The result may be simplified by removing dead leaves (i.e. leaves which are governed by incompatible predicates) and by applying the rule:

$$\textbf{if } p\textbf{ then } x\textbf{ else } x \equiv x. \quad (21)$$

The computation of (15) above was an example of the use of these rules, with:

$$\mathbf{T} = \textbf{if } (i \geq 1 \wedge j < n)\textbf{ then } (s_2, i+1, j-1)\textbf{ else } \perp,$$

and

$$(s_n, \mathbf{K}_n(\mathbf{b})) = (s_1, i+j).$$

One first applies (19) to get:

$$\begin{aligned} \mathbf{S} \;=\; & \textbf{if } (i \geq 1 \wedge j < n) \\ & \textbf{then } \max\nolimits_{\prec}\{(s_2, i+1, j-1), (s_1, i+j)\} \\ & \textbf{else } \max\nolimits_{\prec}\{\perp, (s_1, i+j)\}. \end{aligned}$$

The first branch of the conditional is computed with the help of (20) and the fact that $(s_1, i+j) \prec (s_2, i+1, j-1)$. The second branch is an instance of (18). The result (15) follows.

A more comprehensive example will be presented later.

3.3.3 Avoiding unnecessary work

While the above algorithm always gives a complete and correct solution, in many cases, it is possible to reduce the amount of work by predicting the value of the sequencing predicate.

Suppose we have found two well defined direct dependences $(s_m, \mathbf{K}_m(\mathbf{b}))$ and $(s_n, \mathbf{K}_n(\mathbf{b}))$, respectively at depth p_m and p_n, for the same reference in operation $(t, \mathbf{b})$. Suppose that the depths are different, and for instance that $p_m < p_n$. From the definitions (4) and (10) it follows that:

$$\begin{aligned}
\mathbf{K}_m(\mathbf{b})[1..p_m] &= \mathbf{b}[1..p_m], & (22)\\
\mathbf{K}_m(\mathbf{b})[p_m+1] &< \mathbf{b}[p_m+1], & (23)\\
\mathbf{K}_n(\mathbf{b})[1..p_n] &= \mathbf{b}[1..p_m],
\end{aligned}$$

and hence:

$$\mathbf{K}_m(\mathbf{b})[1..p_m] = \mathbf{K}_n(\mathbf{b})[1..p_m] = \mathbf{b}[1..p_n]. \quad (24)$$

Now, all structured languages have the following property: given two loops, either they have disjoint bodies, or one of them includes the other. In our case, there are loops at depth p_m which include s_m and t, and s_n and t. The bodies of these loops cannot be disjoint, and, since they have the same depth, they are identical. This is tantamount to saying that:

$$N_{s_m s_n} \geq p_m \quad (25)$$

Consider now the sequencing predicate:

$$\begin{aligned}
(s_m, \mathbf{K}_m(\mathbf{b})) \prec (s_n, \mathbf{K}_n(\mathbf{b})) \;\equiv\;& \mathbf{K}_m(\mathbf{b})[1..N_{s_m s_n}] \ll \mathbf{K}_n(\mathbf{b})[1..N_{s_m s_n}]\\
\vee\;& (\mathbf{K}_m(\mathbf{b})[1..N_{s_m s_n}] = \mathbf{K}_n(\mathbf{b})[1..N_{s_m s_n}] \wedge T_{s_m s_n}).
\end{aligned}$$

When evaluating this formula, there are two cases. Firstly, (25) may be strict. From (23) we deduce that the first disjunct is true. If (25) in fact is an equality, then the first disjunct is false[1] and the value of the sequencing predicate simply is $T_{s_m s_n}$. In both cases, we may compute the sequencing predicate without any reference to the actual values of the direct dependences. This result may be used in at least three ways:

[1]Remember that $\ll$ is the *strict* lexical order.

- When computing the direct dependence, use of (24) allows one to reduce the number of unknowns in the parametric integer problem[2].
- When evaluating (20), there is no need to expand the sequencing predicate unless both dependences are at the same depth.
- Most importantly, before embarking on the evaluation of (16), one may check whether $(s_n, \mathbf{K}_n(\mathbf{b}))$ occurs earlier than all leaves of $\mathbf{S}_{n-1}$ or not. In the first case, the evaluation of $\mathbf{K}_n(\mathbf{b})$ is useless. One easily sees that this situation is most likely to occur if the candidate list is ordered by decreasing depth.

3.4 Summary

Let us summarize the algorithm. For a given reference to an array or scalar $\mathbf{M}$ in a statement s, construct the candidate list from all pairs $\langle r, p\rangle$ where r is a statement which modifies $\mathbf{M}$ and p, $0 \le p \le N_{rs}$, is the dependence depth. Set $\mathbf{S} = \perp$. Order the candidate list by decreasing depth.

For each candidate, test if there is a possibility that it will contribute to the final source function. If not, discard the candidate. Otherwise, compute the direct dependence by applying the PIP algorithm to (10). Use (18), (19) and (20) to update the value of the source function and simplify.

The algorithm may appear to be highly complex; there are, however, techniques to reduce the amount of work involved. Most of the time, the algorithm will be embedded in a restructuring compiler[20], which will start by computing the dependence graph of the program. In fact, there is a flow dependence between statements r and s at depth p if the set $\mathsf{Q}^p_{rs}(\mathbf{b})$ is not empty for some legal value of $\mathbf{b}$. Conversely, if there is no dependence, $\mathsf{Q}^p_{rs}(\mathbf{b})$ is empty,

$$\mathbf{K}^p_{rs}(\mathbf{b}) = \perp,$$

and the value of $\mathbf{S}$, as computed by (16), does not change. Hence the only candidates to be considered are those which correspond to flow dependence

[2]Note that in the favourable case when there are no unknowns left, one still has to use the PIP algorithm to check that the obvious solution meets the inequalities constraints of (10).

edges. There are fast approximate methods for the calculation of dependences [28], and more precise methods[26] which are still faster than a PIP computation.

Scalar references are analysed in the same fashion as array references, the only difference being that the index equations $\mathbf{f}_i(\mathbf{u}) = \mathbf{g}(\mathbf{b})$ in (9) now disappear. At first glance, this may be thought of as an important simplification. We have found, in fact, that directly expressing the solution without the help of the PIP algorithm is highly complicated: for instance, one cannot simply say that the latest execution of a loop is the one that correspond to the loop upper limit, since the loop may not be executed at all. As a consequence, we use the general algorithm whatever the rank of the reference.

4 Applications

4.1 Conversion to Single Assignment Form

Single assignment programs have been proposed by several authors[27, 25] as a mean of specifying algorithms for highly parallel systems. Another point[6, 7] is that since a memory cell in such a program is defined only once, its contents may be considered as a "variable" in the mathematical sense and subjected to the familiar algebraic and analytic manipulations.

The following algorithm may be used to convert a static control program to single assignment form:

1 Compute the source function for all rhs references;

2 For each statement s , declare a new array $\mathbf{M}_s$ and replace the left hand side of s by $\mathbf{M}_s[\mathbf{a}]$, where $\mathbf{a}$ is the iteration vector of s;

3 Replace all rhs references by the corresponding source function with the following modifications :

   - replace a leaf of the form $(s, \mathbf{l}(\mathbf{b}))$ by $\mathbf{M}_s[\mathbf{l}(\mathbf{b})]$,
   - replace a void leaf $\perp$ by the original rhs reference.

To justify the last prescription, note that a void source indicates that the corresponding memory cell has not been defined anywhere in the program. As a consequence, its value still is the one it had at the program start.

```
for i := 1 to n do
begin
  for j := 1 to i-1 do
     for k := i+1 to n do
        u[j,k] = u[j,k]-a[i,k]*u[j,i]/u[i,i];  {S1}
  for j := i+1 to n do
     for k := i+1 to n do
        u[j,k] = u[j,k]-u[i,k]*u[j,i]/u[i,i];  {S2}
 end
```

Figure 8: A version of the Gauss-Jordan algorithm

The result of this transformation may be presented as a set of recurrence equations, with all a priori sequencing left out.

Consider for instance the version in figure 8 of the Gauss-Jordan elimination algorithm (declarations and input/output statements omitted). Let us first detail the computation of the source of `a[j,k]` in s_1. s_1 and s_2 both are possible sources. Hence, there will be two direct dependences. A standard dependence analysis will show that all dependences are at depth 0. As a consequence, there are only two candidates, which are given by the PIP algorithm:

$$\mathbf{K}_1 = \textbf{if } i - j \geq 2 \textbf{ then } (i-1, j, k) \textbf{ else } \perp, \tag{26}$$

$$\mathbf{K}_2 = \textbf{if } j \geq 1 \textbf{ then } (j-1, j, k) \textbf{ else } \perp. \tag{27}$$

The problem is now to evaluate recurrence (16). Obviously:

$$\mathbf{S}_1 = \textbf{if } i - j \geq 2 \textbf{ then } (s_1, i-1, j, k) \textbf{ else } \perp.$$

The first step in computing $\mathbf{S}_2$ is to apply rules (19), (18) and (20) to obtain the interim result:

$$
\mathbf{S}_2 = \begin{array}{lll}
\textbf{if} & i - j \geq 2 & \\
\textbf{then} & \textbf{if} & j \geq 2 \\
& \textbf{then} & \begin{array}{ll} \textbf{if} & (s_1, i-1, j, k) \prec (s_2, j-1, j, k) \\ \textbf{then} & (s_2, j-1, j, k) \\ \textbf{else} & (s_1, i-1, j, k) \end{array} \\
& \textbf{else} & (s_1, i-1, j, k) \\
\textbf{else} & \textbf{if} & j \geq 1 \\
& \textbf{then} & (s_2, j-1, j, k) \\
& \textbf{else} & \perp
\end{array}
$$

Examination of the original program gives:

$$
(s_1, i-1, j, k) \prec (s_2, j-1, i, k) \equiv i - 1 \leq j - 1
$$

which is false when $i - j \geq 2$: this is a case of elimination of a dead leaf. Next comes an application of (21), and the final result is:

$$
\mathbf{S}_2 = \begin{array}{lll}
\textbf{if} & i - j \geq 2 & \\
\textbf{then} & (s_1, i-1, j, k) & \\
\textbf{else} & \textbf{if} & j \geq 1 \\
& \textbf{then} & (s_2, j-1, j, k) \\
& \textbf{else} & \perp
\end{array}
$$

Similar calculations for all other rhs references gives the LAU form of figure 9. This result is quite involved, and may be simplified in several ways. However, we do not advocate that such a code be used for actual computing, but rather as a starting point for further analysis and optimization. Hence, simplification per se may not be worth the effort.

4.2 Array and Scalar Expansion

Parallel or vector execution of a program may be frustrated by allocation of the same memory cell to unrelated values. This is called an output dependence[20]. Transforming the program to single assignment style removes all such dependences, at the cost of a large increase in memory usage.

$$1 \le i \le n, 1 \le j \le i-1, i+1 \le k \le n :$$

```
u1[i,j,k] = if (i-j-2 >= 0) then u1[i-1,j,k]
             else if (j-2 >= 0)
                  then u2[j-1,j,k]
                  else u[j,k]
            - u2[i-1,i,k] / u2[i-1,i,i] *
                       if (i-j-2 >= 0) then u1[i-1,j,i]
                        else if (j-2 >= 0)
                             then u2[j-1,j,i]
                             else u[j,i]
```

$$1 \le i \le n, i+1 \le j \le n, i+1 \le k \le n :$$

```
u2[i,j,k] = (if i-2 >= 0 then u2[i-1,j,k] else a[j,k])
          - (if i-2 >= 0 then u2[i-1,i,k] else a[i,k])
          * (if i-2 >= 0 then u2[i-1,j,i] else a[j,i])
          / (if i-2 >= 0 then u2[i-1,i,i] else a[i,i])
```

Figure 9: The single assignment form of program 8

In many cases, such expansion is useless and should not be done. For instance, on most vector computers, innermost loops are the only ones which are susceptible of vector mode execution. In other cases, the output dependence is accompanied by a true dependence, which cannot be eliminated by expansion. The problem of deciding which lhs should be expanded and/or renamed is highly dependent on the target computer and will not be addressed here. We will suppose that we are given a list of modified lhs, the new lhs for operation $(s, \mathbf{a})$ being $\mathbf{M}_s[\mathbf{f}(\mathbf{a})]$. Most often, $\mathbf{f}$ will be a selection operator on the components of $\mathbf{a}$. One then applies the algorithm of Section 4.1, with step 3 modified in the following fashion :

3' Replace all rhs references by the corresponding source function with the following modifications :

- replace a leaf of the form $(s, \mathbf{l}(\mathbf{b}))$ by $\mathbf{M}_s[\mathbf{f}(\mathbf{l}(\mathbf{b}))]$ if the lhs of s has been modified, and by the original rhs if s is untouched.
- replace a void leaf $\perp$ by the original rhs reference.

Obviously, a rhs all of whose sources are untouched is not modified by the above prescription.

Note that not all renaming and expansion are legitimate. When one needs a value, one must take care that it has not been overwritten some time before. There is a precise solution to this problem. To check that a value produced by $(s, \mathbf{K}(\mathbf{b}))$ with lhs $\mathbf{M}_s(\mathbf{f}(\mathbf{a}))$ is still available at $(t, \mathbf{b})$, one should test that for all statements r with lhs $\mathbf{M}_s[\mathbf{h}(\mathbf{c})]$ the following problem :

$$\begin{aligned} \mathbf{h}(\mathbf{c}) &= \mathbf{f}(\mathbf{b}), \\ (s, \mathbf{K}(\mathbf{b})) &\prec (r, \mathbf{c}) \prec (t, \mathbf{b}), \\ \mathbf{e}_r(\mathbf{c}) &\geq 0, \end{aligned}$$

has no solution in $\mathbf{c}$ in the context $\mathbf{e}_t(\mathbf{b}) \geq 0$. There are many cases in which this calculation is not necessary. Let us note the case in which $\mathbf{M}_s$ is used only in s, and the one in which all uses of $\mathbf{M}_s$ have as indices a superset of the indices of the original lhs.

4.3 Program Checking and Optimization

Here we will suppose that we are given a program complete with initializations and input/output statements. These statements are easily included in the present framework. For instance, an output statement may be modelled as a statement with rhs references but no lhs. The first step in the verification of such a program is to check the sources for the presence of the $\perp$ sign, which indicates access to an undefined memory cell.

When computing a source, one may refine the polyhedron $\mathsf{Q}(\mathbf{b})$ by adding linear constraints expressing the fact that all indices are within the array bounds. The $\perp$ sign will in that case pin-point an out-of-bound access. Most often, the $\perp$ sign will appear inside a conditional whose predicate gives a condition on the structure parameters which must be checked for the program to run correctly. Adding a run-time test for this condition is a simple matter.

Knowledge of the source functions allows very precise detection of dead code. Certainly all output statements are useful code and should be marked accordingly. If statement t is marked, all statements which occur in sources for rhs references in t are useful. When this process (which is nothing more than a graph traversal algorithm) terminates, unmarked statements are dead code.

Finally, the single assignment form of a program is an invaluable help in checking that the program has the desired behaviour. Consider for instance two very similar pieces of code:

```
for i:= 1 to n do a[i] := a[i+1]   {1}

for i:= 1 to n do a[i] := a[i-1]   {2}
```

Their single assignment transcriptions are widely different:

```
for i:= 1 to n do A[i] := a[i+1] {1}

for i:= 1 to n do
   A[i] := if i-1 >= 0  {2}
           then A[i-1]
           else a[i-1]
```

where `A` is a new array.

In the case of $\{2\}$, the assignment :

```
   A[i] := A[i-1]
```

may be considered as a recurrence in the usual mathematical sense and solved to yield :

```
   A[i] = a[0]
```

4.4 Parallel Program Construction

An obvious idea is to summarize the source function by a graph. There is an edge from s to t for each occurence of s in a source of a rhs reference in t. This gives the dataflow graph of the original program. It is obtained from the usual dependence graph[20] by removing output dependences, anti-dependences and spurious flow dependences. This graph may be submitted to classical parallelization and vectorization algorithms[2]. One still has to expand some variables to reconstruct a correct program.

Another approach is to consider the source functions as synchronization constraints (a statement which uses a given value may not start executing until the source statement has terminated), and to attempt the construction of a parallel program which meets all of them. This approach is reminescent of the methodology for the automatic or semi-automatic design of systolic arrays [22], and leads to the consideration of timing functions or schedules. The use of timing functions for the construction of parallel program has been advocated in several papers[13, 21, 19]. The outcome of this research will be reported elsewhere.

5 Related Work

This paper is related to work in two different areas: one is standard dataflow analysis[1], which is used as a basic technique by many optimizing compilers, and the other is the specification and compilation of dataflow languages.

Standard dataflow analysis is both more and less comprehensive than the present one. Its range of applicability is wider, since it deals with unstructured programs. However, it is a static theory (all executions of a statement in a loop are lumped as one), and, as such, applies only to scalars (or to arrays considered as a whole). An example is the determination of use-def chains. 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. In our framework, use-def chains could be obtained by computing the frontier of the source functions and removing all informations about iteration vectors.

In a similar context, a technique for conversion to static single assignment form has been advocated by Cytron et. al. [11]. Here again, the source program is not required to be structured, and only scalars or arrays taken as a whole are considered. The paper is concerned with the most economical insertion of so-called ϕ-functions (i.e. multiplexors) at join points in the control graph. When this is done, it is possible to rename all variables and to obtain a single assignment program.

Dataflow architectures are one of several ways of exploiting single assignment programs. Each architecture has a machine language, which in general is presented as a dataflow graph. One of the problems in this field is how to provide a more user-friendly interface, either in the form of a high-level parallel language, or by translating conventional language to dataflow. Our work is certainly relevant to this aim. A recent paper[9] gives an algorithm for translating FORTRAN to dataflow graphs. Here again the problem is with arrays. A dataflow machine has no difficulty in executing the flowgraph equivalents of `doall` or `doacross` loops. Detecting such loops, however, must use classical techniques like dependence analysis.

Dependence analysis is mainly used by parallelizing and vectorizing compilers. There is a flow or true dependence between two statements if the first one is a possible source for a value which is used by the other[20, 28]. There are other kinds of dependences: anti- and output-dependences, which indicate memory sharing, and control dependences, which summarize the control flow in the source program.

One may say that a flow dependence is a very imprecise approximation to the source function. Some more precise descriptions are the dependence direction vectors[28], the dependence vectors[18], the dependence cone[15] and the direct dependences[10].

Scalar expansion[20] is the particular case of the present problem in which the modified variable is a scalar which is expanded to a vector. If one restricts oneself to innermost loops, the problem has a very simple solution.

6 Conclusion

The main result of this paper is the description of an algorithm for the dataflow analysis of programs with array references and `for` loops. It has been implemented partly in Lisp and partly in C, and runs on several computers ranging from a personal computer to a DEC Vax 11/780. No effort has been made (at the time of writing) to optimize the code (the Lisp to C interface

	lines	lhs	rhs	level	leaves	CPU
across	10	4	5	1	8	0.6
burg	27	11	20	2	40	5.6
relax	11	1	4	3	10	1.7
gosser	19	5	11	3	20	2.8
choles	21	6	8	3	12	2.6
lanczos	55	23	31	3	54	12.6
jacobi	50	31	60	4	92	81.9

Table 1: Some kernels and their dataflow analysis

is especially clumsy).

Table 1 gives some quantitative results for a set of small to medium kernels. For each program we give the line count, the number of assignment statements (lhs), the number of rhs references and the maximum nesting level. The results are the number of leaves in the source quasts (which caracterizes the complexity of the solution), and the CPU time in seconds on a low-end SPARC station. One may observe that the source functions are quite simple: about two leaves per rhs reference. As to the CPU time, the main controlling factor seems to be the maximum nesting level in the program. The time per leaf goes from 75 ms for a one level program to 890 ms for a four level program. While these values may be somewhat reduced by converting the Lisp part of the program to C, we do not expect more than one order of magnitude improvement. It seems clear that the method will be applied only to small kernels or to larger programs whose running time is highly critical (e.g., library modules).

We have described several applications of our technique; the reader will probably be able to add several new items to the list. Most of these are especially interesting in the context of automatic parallel program construction and will be developed with this kind of application in mind. Some of these methods will require further study to become operational; these unsolved points have been noted where appropriate.

Extending the technique to languages with fewer restrictions than we introduced in section 2.2 would be highly interesting. Some estimate of the applicability of our technique may be deduced from the statistics of Zhiyu

Shen et. al.[23]. The main difficulty is non-linear indices. In this paper, which analyses more than 100 000 lines of code, about 53% of all indices are found to be linear, about 13% are partially linear, and the remaining 34% are non-linear. An index is classified as partially linear as soon as it contains a variable which is not a loop counter. Some of these unknown variables may be eliminated by forward substitution. Some others are structure parameters. Hence we expect that the only significant failure cause will be the use of an array element as an index, which account for about 7% of all cases.

Before being submitted to a dataflow analysis, a program must be put in structured form. There are technique for the elimination of `goto`'s[8] and for the detection of induction variables[1], which then allows one, in favourable cases, to reconstruct unit increment `for` loops and to delete extraneous variables by forward substitution[5, 4, 16].

We expect that the handling of conditionals (by the familiar device of reducing them to guards on assignment statements) would not be too difficult. Conditionals whose predicate depends only on loop counters should be handled as restriction on the iteration domains of the controlled statements. `while` loops may perhaps be handled, in the context of parallel program construction, as loops with an unbounded iteration domain. On the other hand, linearity restrictions are crucial for the applicability of the method, and we do not envision at the present time any trick for dispensing with them.

Lastly, the analysis of programs with procedure and function calls is a very difficult problem. If we restrict ourselves to the handling of small kernels, a few tricks should do the job: identify those function calls which act as operators (no argument is modified, no global variable is accessed). Other subroutine calls should probably be inlined.

7 Acknowledgments

This work has been supported by DRET under contract 87/280 and by PRC C^3 of the french CNRS. Part of section 2.3 has been reproduced[12] by permission of ACM.

A The Parametric Integer Algorithm

A.1 The Basic Algorithm

A parametric integer program (PIP) may be formulated in the following way. Let $\mathsf{F}(\mathbf{z})$ be the set of integer points inside a convex polyhedron:

$$\mathsf{F}(\mathbf{z}) = \{\mathbf{x} | S\mathbf{x} + \mathbf{t}(\mathbf{z}) \in \mathsf{N}\} / K\mathbf{z} + \mathbf{h} \in \mathsf{N}, \tag{28}$$

where S and K are matrices and $\mathbf{t}(\mathbf{z})$ is an integer vector whose components are affine functions of the integer vector $\mathbf{z}$. $\mathbf{z}$ is constrained by the set of inequalities

$$K\mathbf{z} + \mathbf{h} \in \mathsf{N},$$

the context of the problem. As a matter of convenience, we will suppose that both S and K are such that they restrict $\mathbf{x}$ and $\mathbf{z}$ to non-negative integer values. In particular, the first $|\mathbf{x}|$ rows of S will generate the constraint $\mathbf{x} \in \mathsf{N}$.

The problem is to decide for which values of $\mathbf{z}$ is $\mathsf{F}(\mathbf{z})$ empty, and if not, to compute its lexical minimum, as a function of $\mathbf{z}$. The solution is given by the following algorithm :

Algorithm N

1 Determine the signs of the components of $\mathbf{t}(\mathbf{z})$ in the context

$$K\mathbf{z} + \mathbf{h} \in \mathsf{N},$$

by solving non-parametric auxilliary integer programs. The sign may be positive, negative or unknown.

2 If there is a negative $\mathbf{t}_i(\mathbf{z})$, then either:

2.1 All elements of $S_{i\bullet}$ are negative. In this case, $\mathsf{F}(\mathbf{z})$ is empty, and the solution is $\perp$.

2.2 There is at least a positive S_{ij}; a pivoting step is executed, giving a new problem $\langle S', \mathbf{t}'(\mathbf{z})\rangle$. The solution of the initial problem is the same as that of the new problem in the old context. In that case, keep track of D, the product of the pivots.

3 If all $\mathbf{t}_i(\mathbf{z})$ are positive, select the earliest row i such that one of S_{ij} or the coefficients in $\mathbf{t}_i(\mathbf{z})$ is not integral. If no such row exists (in particular if $D = 1$), the solution has been found; it is given by the first $|\mathbf{x}|$ components of $\mathbf{t}(\mathbf{z})$. If such a row exists, let q be a new parameter. Add :

$$0 \leq ((-D\mathbf{t}_i(\mathbf{z})) \bmod D) - qD \leq D - 1,$$

to the context. Let m be the number of rows in S. Add to S the new row $m+1$ with the following coefficients :

$$\begin{aligned} S_{(m+1)j} &= ((DS_{ij}) \bmod D)/D, \\ \mathbf{t}_{m+1}(\mathbf{z}) &= (-((-D\mathbf{t}_i(\mathbf{z})) \bmod D)/D) + q, \end{aligned}$$

and start again at step 1.

4 In the remaining case, select a $\mathbf{t}_i(\mathbf{z})$ whose sign is unknown; let x_+ and x_- be respectively the solutions of $\langle S, \mathbf{t}(\mathbf{z})\rangle$ in the contexts

$$K\mathbf{z} + \mathbf{h} \in \mathsf{N}, \mathbf{t}_i(\mathbf{z}) \geq 0,$$

and

$$K\mathbf{z} + \mathbf{h} \in \mathsf{N}, \mathbf{t}_i(\mathbf{z}) < 0,$$

respectively. The solution of the initial problem is :

$$\mathbf{if}\ \mathbf{t}_i(\mathbf{z}) \geq 0\ \mathbf{then}\ x_+\ \mathbf{else}\ x_-.$$

This algorithm is guaranteed to terminate (see [14]). The result is a multilevel conditional expression whose predicates and leaves are affine functions of the parameters. The new parameters like q above may be replaced by their expressions as integer quotients of affine forms.

The algorithm above is not entirely deterministic; there are many equivalent solutions to the same problem. Experience has shown that a few simple heuristics suffice for selecting a well behaved solution. First of all, avoid splitting (case 4) at all cost (e.g. by grouping the case $\mathbf{t}_i(\mathbf{z}) = 0$ with the positive or negative case if the other case does not exist). If forced to split, select a row with all coefficients negative, which implies that $x_- = \perp$. This algorithm has been implemented both in Lisp and C; these codes have been used to run all examples in this paper.

A.2 The lexical maximum

In many cases of interest, one has to compute a lexical maximum rather than a minimum. Sometimes, a transformation from one problem to the other is in evidence. We favour, however, the following systematic procedure.

Algorithm M

Refering back to (28), introduce a new "very large" parameter m and solve:

$$\mathbf{u} = \min_{\ll} \mathsf{G}(\mathbf{z}, m)/K\mathbf{z} + \mathbf{h} \in \mathsf{N},$$

where

$$\mathsf{G}(\mathbf{z}, m) = \{\mathbf{y} | 0 \le \mathbf{y} \le m, -S\mathbf{y} + S\mathbf{1}m + \mathbf{t}(\mathbf{z}) \in \mathsf{N}\}.$$

Compute[3] $\mathbf{v} = m\mathbf{1} - \mathbf{u}$ and prune the solution by replacing all tests in whose predicate m has a positive coefficient by their true branch and conversely. A leaf in which m occurs with a positive coefficient is associated to a range of the parameters where $\mathsf{F}(\mathbf{z})$ is unbounded. This case will never occur in the problems we are interested in.

It is easy to prove that $\mathbf{v}$ is the required maximum; it is also easy to devise methods to do the pruning "on line", so as to keep the extra computation to a minimum. For instance, in step (1) of algorithm N, if m occurs with a positive sign in $\mathbf{t}_i(\mathbf{z})$, the i-th line may be taken as positive. We have found in practice that in cases where we need to compute both the maximum and the minimum of the same set, both algorithms have operation counts of the same order of magnitude, and neither of them is systematically longer than the other.

References

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

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

[3] $\mathbf{1}$ is the vector all of whose components are 1.

[3] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM TOPLAS*, 9(4):491–542, October 1987.

[4] Zahira Ammarguellat. *Normalization of Program Control Flow.* Technical Report 885, CSRD, May 1989.

[5] Zahira Ammarguellat. *Restructuration des programmes FORTRAN en vue de leur parallélisation.* PhD thesis, Université P. et M. Curie, Paris, December 1988.

[6] Jacques Arsac. *La construction de programmes structurés.* Dunod, Paris, 1977.

[7] E. A. Ashcroft and W. W. Wadge. *Lucid, the Data-flow Programming Language.* Academic Press, 1985.

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

[9] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[10] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *ACM Int. Conf. on Supercomputing*, St Malo, France, July 1988.

[11] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proc. 16th ACM POPL Conf.*, pages 25–35, January 1989.

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

[13] Paul Feautrier. Asymptotically efficent algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized Systems*, pages 273–284, IFIP WG 10.3, North-Holland, December 1989.

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

[15] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.

[16] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Procs. of the 3rd Int. Conf. on Supercomputing*, pages 186–194, ACM Press, 1989.

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

[18] Leslie Lamport. The parallel execution of DO loops. *CACM*, 17:83–93, February 1974.

[19] Lee-Chung Lu. A unified framework for systematic loop transformations. *SIGPLAN Notices*, 26:28–38, July 1991. 3rd ACM SIGPLAN Symp. on Priciples and Practice of Parallel Programming.

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

[21] William Pugh. Uniform techniques for loop optimization. *ACM Conf. on Supercomputing*, 341–352, January 1991.

[22] Patrice Quinton. Mapping recurrences on parallel architectures. In *3rd Int. Conf. on Supercomputing*, Boston, May 1988.

[23] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study on array subscripts and data dependencies. In *1989 Int. Conf. on Parallel Processing*, pages II 145–152, 1989.

[24] N. Suzuki and D. Jefferson. Verification decidability of Pressburger array programs. In *Procs. of a conf. on TCS*, Waterloo, 1977.

[25] J.C. Syre, D. Comte, and N. Hifdi. Pipelining, parallelism and asynchronism in the LAU system. In *Int. Conf. on Parallel Processing*, 1977.

[26] Nadia Tawbi, Alain Dumay, and Paul Feautrier. *PAF : un paralléliseur automatique pour FORTRAN.* Technical Report 185, MASI, 1987.

[27] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *SJCC*, pages 403–408, 1968.

[28] Michael J. Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *Int. J. of Parallel Processing*, 16:137–178, 1987.

```
@ARTICLE{Feau:91,
        AUTHOR = "Paul Feautrier",
        TITLE = "Dataflow Analysis of Scalar and Array References",
        JOURNAL = "Int. J. of Parallel Programming",
        VOLUME = 20, NUMBER = 1,
        YEAR = "1991", MONTH = Feb, PAGES = "23--53"
}
```