Array Dataflow Analysis

Paul Feautrier *

February 8, 2006

May 21, 1997

Abstract

While mathematical reasoning is about fixed *values*, programs are written in term of *memory cells*, whose contents are changeable values. To reason about programs, the first step is always to abstract from the memory cells to the values they contains at a given point in the execution of the program. This step, which is known as Dataflow Analysis, may use different techniques according to the required accuracy and the type of programs to be analyzed.

This paper gives a review of the *ad hoc* techniques which have been designed for the analysis of Array Programs. An exact solution is possible for the tightly constrained *static control programs*. The method can be extended to more general programs, but the results are then approximation to the real dataflow. Extensions to complex statements and to the interprocedural case are also presented.

The results of Array Dataflow Analysis may be of use for program checking, program optimization and parallelization.

1 Introduction

There are many situations in which one needs to thoroughly understand the behavior of a program. The most obvious one is at program checking time. If we could extract a description of a program as, e.g., a set of mathematical equations and compare it to a specification, also given in the same medium, debugging would become a science instead of an art. Reverse engineering is another case in point. But the most important application of such analyses is to optimization. Each optimization has to be proved valid in the sense that it does not modifies the program ultimate results. To achieve this, we have to know, in a more or less precise way, what these results are intended to be. Since the most aggressive type of optimization a program can be subjected to is parallelization, understanding a program before attempting to parallelize it is a very important step.

Now, since the time of Von Neuman, programs are written in term of "variables" which are in fact symbolic names for memory cells. Values are never given[1], or even named, but always alluded to as "the present content of cell x". On the other hand, in mathematics, the subject of discourse is always a

*e-mail : `Paul.Feautrier@ens-lyon.fr`

[1] except in the case of constants.

value which never change, albeit it can be unknown or arbitrary. The value in a given memory cell can be modeled as a function of time (that function may be constant).

Obviously, "time" here is not physical time. Besides the fact that exhibiting such a function would be nearly impossible, it would have the added inconvenience of not being portable among different computers. We will use a logical time, to be defined later. The only requirement is that there must be a "time arrow": time must belong to an ordered set. Since the state of a computer memory does not change except at each execution of an assignment, logical time is not continuous but discrete. Each time step is an *operation* of the computer, which corresponds, from the point of view of the programmer, to the execution of an instruction. For program analysis purposes, there is some leeway in the definition of an operation. It may be the execution of a machine instruction, as in the case of Instruction Level Parallelization, or the execution of an assignment statement, as in most of this paper, or the execution of a complex statement, as in Sect. 4.

If we stipulate that the meaning of a program is given by expressing the value of variables as a function of (logical) time, then dataflow analysis is the process of extracting properties of these functions from the program text. These properties may be of widely varying precision. In some cases, one may exhibit a closed formula for the function. In other cases, one may only knows that it has positive values. In the most frequent cases, one has to be content with *relations* between values taken either at the same time (Floyd's assertions [Flo67]) or at different times. As before, these relations may be more or less precise. We will show that, for a simple but useful category of programs, the result of Array Dataflow Analysis is a system of *equations* relating the values of variables at distinct time points.

Dataflow analysis is based on the observation that the value one may retrieve from a memory cell is the one which was written last. In the scalar case, this allows one to write *dataflow equations*, which may be solved either by iterative methods or by direct methods. In the case of array cells, the problem is more difficult because there is no simple method for deciding if two references to the same array are references to the same cell or not: two occurrences of `a[i]` are references to the same cell iff `i` has not been modified in between. Conversely, it may happen that `a[i]` and `a[j]` refer to the same cell if the values i and j[2] are equal.

There is a general method for devising dataflow analyses [CC77]. One starts from a semantical description of the source language, and then one abstracts the features of interest by constructing a nonstandard semantics. The result of executing a given program according to this semantics, if possible, is the required property.

Our main interest here is another type of analysis which has been designed in an *ad hoc* way for the use of automatic parallelizers. The initial concept was that of *dependences*. There is a flow dependence between statement $\mathtt{S}_1$ and $\mathtt{S}_2$ iff a value produced by $\mathtt{S}_1$ may be used later by $\mathtt{S}_2$. By restricting the allowed expressions in subscripts and loop bounds to affine expressions, the problem reduces to the question of the feasibility in integers of a system of affine

[2] We will adhere to the following convention: identifiers will always be written in a `Teletype` font. Their values at a given time will always be denoted by the same letter in an *italic* font. If necessary, the time will be indicated by various devices (accents, subscripts, arguments).

inequalities. The problem is solved by standard Linear Integer Programming algorithms. It was soon realized [Fea88a] that the same technology could give much more precise results. For programs abiding to the same restrictions as above, and for each value in the program, one can pinpoint its *source*, i.e. the name of the write operation which created it. This information is invaluable for program checking, program understanding (a.k.a. reverse engineering), program optimization and parallelization.

Program whose only control structure is the `do` loop, whose only data structure is the array and in which loop bounds and subscripts are affine functions are known as *static control programs*. For such programs, one can take iteration vectors (the vectors whose components are the current values of the loop counters) as logical time. It follows that, under the above hypotheses, array subscripts are closed functions of (logical) time. This is the crucial property which allows us to find relations between the other values in the program. For program which are outside the static control model, devising an Array Dataflow Analysis is much more difficult. A first possibility is to extend slightly the control model by adding conditionals and `while` loops. If this is not possible, it means that the iteration count of the loop cannot be bounded at compile time. The consequence is that, if these iterations can be the source of a value, then we cannot find the last one. In that case, all we can do is to report that the source belongs to a set of iterations. The result of our analysis is no longer sources, but source sets, and our aim will be to find the smallest possible source *sets*. The corresponding technique is known as Fuzzy Array Dataflow Analysis and is presented in Sect. 3. It can be extended to the case where some subscripts are no longer affine functions [BCF97].

We will next present some extensions of ADA. The first one is to statements which may return an unbounded number of results. Typical cases are `read` statements, vector statements à la Fortran 90 and `forall` statements à la HPF (Sect. 4). Procedures may return an unbounded number of results as soon as they have at least one array argument. Hence, they belong to the above category and can be treated in the same way as, e.g., vector operations. These techniques are presented in Sect. 4.3.

In the conclusion, we sketch some applications of Array Dataflow Analysis and point to several unsolved problems.

The work which is reported here has taken many years of research by many peoples to evolve from the rough sketch in [Fea88a] to the present state of affairs. I would like to acknowledge contributions by Denis Barthou and Jean-François Collard [BCF97], by Vincent Lefebvre [LF97] and by Arnauld Leservot [Les96].

2 Exact Array Dataflow Analysis

Exact Array Dataflow Analysis is possible only in the case of static control programs. We will first describe this program model. The results of exact ADA are *source functions*, which give, for each step in the execution of the source program and for each memory cell, the operation which has generated the current value of the memory cell. We give an algorithm for computing source functions and compare it to other proposals from the literature.

2.1 Notations

The objects we have to handle in this paper are mainly vectors with integer coordinates and set of such vectors. $|\vec{a}|$ is the dimension of $\vec{a}$. $\vec{a}[i..j]$ is the subvector of $\vec{a}$ built from components i to j. $\vec{a}[i]$ is a shorthand for $\vec{a}[i..i]$. Familiar operators and predicates like $+$ and $\geq$ will be tacitly extended to vectors. The sign $\ll$ denote lexical ordering of vectors. The max operator, when acting on vectors or vector sets, is always to be understood as the maximum according to $\ll$. Large letters will usually denote sets; $\mathbb{N}$ will be the set of nonnegative integers and $\mathbb{Z}$ the set of signed integers.

2.2 The Program Model

Let us first insist that the present work is not about any particular language, but about the static subset of any programming language. To emphasize this fact, the examples will be written indifferently in Fortran, Pascal or C. Furthermore, the fact that a given program fragment belongs to this static subset may be self-evident from the program text, or may be the result of elaborate preprocessing (`goto` elimination, induction variable detection, constant propagation, `do` loop reconstruction, to cite a few). In this paper, we will always suppose that such preprocessing has already been applied and that we are dealing with its results.

For simplicity, data types will be restricted to integers, reals, and n-dimensional arrays of integers and reals. Adding other scalar types (Boolean, complex numbers) and even record types is easy. The only statements we will consider in this section are scalar and array assignments. The only control constructs will be the sequence and the `do` loop. A `do` loop has the property that it possesses a counter, and that neither the counter nor its upper and lower bounds are modified by the loop body. In this paper, we will suppose that the loop step is always one. If the step is a known numerical constant, the program can always be transformed to have step one. If the step is an expression, the program will be considered to be beyond the static control model.

> The Pascal `for` loop has all of the above properties and thus can be considered equivalent to a Fortran `do` loop. The C `for` loop is a more complex object since the loop counter, lower and upper bounds are not recognized by the language, and since these elements can be modified in the loop body. However, it is possible to check whether these restriction are adhered to, and thus to identify those C loops which are equivalent to a Fortran loop.

We will also suppose that compound statements are *flattened*, i.e. that constructions such as

```
begin S1;
  begin S2; S3
  end
end
```

are replaced by the equivalent:

```
begin S1; S2; S3 end
```

2.2.1 Restrictions

The above restrictions are obviously intended to simplify the calculation of the total number of iterations of all loops. This is, however, not sufficient: we have to specify the form and content of the loop bounds. 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). These observations motivate the following definition of the class of static control programs.

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 `do` loops whose bounds depend only on structure parameters, numerical constants and outer loops iteration counters. The analysis technique which is presented here is based on the theory of affine inequalities, and hence is applicable only if all limits are affine functions. For similar reasons, all subscripts are restricted to affine functions of the loop counters and the structure parameters.

We will use the fact that in a correct program, array subscripts 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 subscripts 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 (see Sect. 5 or, for a precursor of our work, [SJ77]).

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 – as for instance Fortran 90.

2.2.2 The Sequencing Predicate

Values in array elements are produced by execution of statements. Hence we need a notation to pinpoint a specific execution of a statement, or *operation*. Our first need is an unambiguous designation of a statement in a program. Our solution is to use arbitrary names, which will be denoted by letters such as $\mathtt{R}$, $\mathtt{S}$, $\mathtt{T}$. When discussing examples, we will use the fact that our preferred languages allow the affixing of a numerical label to each statement. By convention, the statement labeled i will be named $\mathtt{S}_i$. 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 fragments, the only repetitive construct is the `do` loop. Hence, an operation is uniquely defined by the name of the statement and the values of the surrounding loop counters (the *iteration vector* [Kuc78]). A pair such as $\langle \mathtt{R}, \vec{a} \rangle$ 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 $\vec{a}$ must be equal to the number of loops surrounding $\mathtt{R}$;
- all components of $\vec{a}$ must be within the corresponding loop limits.

With each loop $\mathtt{L}$ we may associate a pair of inequalities:

$$lb_{\mathtt{L}} \leq a \leq ub_{\mathtt{L}},$$

where a is the loop counter of $\mathtt{L}$. If a statement $\mathtt{R}$ is embedded in a loop nest $\mathtt{L}_1, \mathtt{L}_2, \ldots, \mathtt{L}_N$, in that order, then the iteration vector $\vec{a}$ of $\mathtt{R}$ must satisfy:

$$\forall p : (1 \leq p \leq N)\, lb_{\mathtt{L}_p} \leq \vec{a}[p] \leq ub_{\mathtt{L}_p}. \tag{1}$$

(1) may be summarized in matrix form as:

$$E_{\mathtt{R}}\vec{a} \geq \vec{n}_{\mathtt{R}}. \tag{2}$$

where $E_{\mathtt{R}}$ is a $2N \times N$ matrix and $\vec{n}_{\mathtt{R}}$ is a vector of dimension N in which the structure parameters may occur linearly.

Formula (2) will be called the existence predicate of $\mathtt{R}$. Notice that we do not suppose that $lb_{\mathtt{L}} \leq ub_{\mathtt{L}}$. In accordance with the Pascal convention (and with the "modern" interpretation of Fortran `do` loops), a loop whose bounds violate this inequality will not be executed at all.

> Consider for example the program sketch in figure 1. Figure 2 describes its iteration domains. The existence predicate of statement $\mathtt{S}_2$ may be written as:
>
> $$\begin{pmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \end{pmatrix} \geq 0.$$

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 [PW86]. Usually, loops are explained from a temporal point of view: iteration i is executed just before iteration $i+1$, and so on. 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:

$$\langle \mathtt{R}, \vec{a} \rangle \prec \langle \mathtt{S}, \vec{b} \rangle.$$

and expresses the fact that $\langle \mathtt{R}, \vec{a} \rangle$ is executed before $\langle \mathtt{S}, \vec{b} \rangle$. The sequencing predicate depends only on the source program text. We have given a simple expression for it in [Fea91]. Let N_{RS} be the number of loops which enclose both statements $\mathtt{R}$ and $\mathtt{S}$. Let $<_{\text{text}}$ be the textual order of the source program: $\mathtt{R} <_{\text{text}} \mathtt{S}$ iff $\mathtt{R}$ occurs before $\mathtt{S}$ in the program text. The execution order is given by:

$$\langle \mathtt{R}, \vec{a} \rangle \prec \langle \mathtt{S}, \vec{b} \rangle \equiv \vec{a}[1..N_{RS}] \ll \vec{b}[1..N_{RS}] \vee (\vec{a}[1..N_{RS}] = \vec{b}[1..N_{RS}] \wedge \mathtt{R} <_{\text{text}} \mathtt{S}). \tag{3}$$

```
DO i =1,n
  DO j = 1,i-1
    S1
  END DO
  DO j = i+1,n
    S2
  END DO
END
```

Figure 1: A sample program

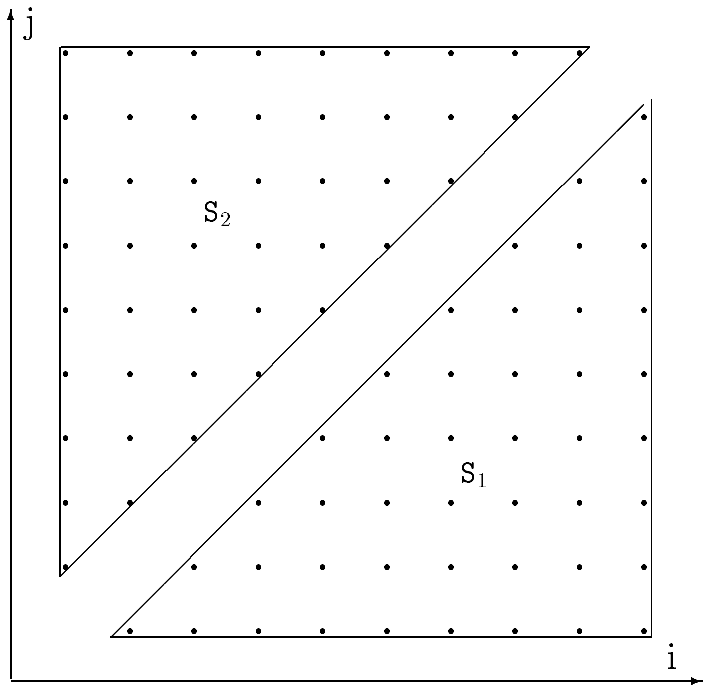

Figure 2: The iteration domain of program 1

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

Figure 3: The product of two polynomials

Knowledge of N_{RS} (a matrix of integers) and $<_{\text{text}}$ (a strict total order relation) 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 $\prec_p$:

$$\langle \mathtt{R}, \vec{a}\rangle \prec_p \langle \mathtt{S}, \vec{b}\rangle \equiv (\vec{a}[1..p] = \vec{b}[1..p] \wedge \vec{a}[p+1] < \vec{b}[p+1]), \quad 0 \leq p < N_{RS}. \tag{4}$$

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

$$\langle \mathtt{R}, \vec{a}\rangle \prec_p \langle \mathtt{S}, \vec{b}\rangle \equiv \vec{a}[1..N_{RS}] = \vec{b}[1..N_{RS}] \wedge \mathtt{R} <_{\text{text}} \mathtt{S}. \tag{5}$$

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[AK87], 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}.

2.2.3 Another Presentation of the Sequencing Predicate

We can derive another expression for the sequencing predicate by considering the *execution tree* of the program, which is obtained by (conceptually) unrolling all its loops. The nodes of the execution tree are either simple statements (the leaves) or compound statements (the interior nodes). A compound statement comes either from a genuine compound statement in the source program or from the unrolling of a loop. Let us number all edges issuing from a given node consecutively from left to right, starting from the lower bound of the loop in the case of unrolling, and from 1 in the case of a compound statement. The coordinates of the iteration vector of a leaf are the numbers associated to the unique path from the root to the leaf in top-down order. If we suppose that the program has been normalized, i.e. that the body of a loop is always a compound statement whatever the number of statements it contains, then the coordinates of the iteration vector alternate between positions in compound statements (constants) and loop counters (variables). By convention, the whole program is a compound statement, hence the the first component of all iteration vectors is a constant. The point of this construction is now that the sequencing predicate is simply lexicographic order.

> Consider the program of Fig. 3. The iteration vectors of $\mathtt{S}_1$ and $\mathtt{S}_2$ are now $\langle 1, k, 1\rangle$ and $\langle 2, i, 1, j, 1\rangle$. From this we deduce, e.g. that all instances of $\mathtt{S}_1$ execute before all instances of $\mathtt{S}_2$. Similarly, by simplifying the lexicographic order, one can show that:
>
> $$\begin{aligned}\langle \mathtt{S}_2, i, j\rangle \prec \langle \mathtt{S}_2, i', j'\rangle &\equiv \langle 2, i, 1, j, 1\rangle \ll \langle 2, i', 1, j', 1\rangle \\ &\equiv i < i' \vee (i = i' \wedge j < j').\end{aligned}$$

The notations we have defined in the preceding section will be extended to deal with the new iteration vectors. For instance, the existence predicate of a statement $\mathtt{S}$ will still be written:

$$E_{\mathrm{R}}\vec{a} \geq \vec{n}_{\mathrm{R}}$$

where the matrix $E_{\mathtt{R}}$ and the vector $\vec{n}_{\mathtt{R}}$ have new rows to deal with the constant values in the iteration vector. Similarly, we will still use $\vec{a} \prec_p \vec{b}$ for the depth p sequencing predicate, the meaning being that the above expression begins by p equalities on the variable components of $\vec{a}$ and $\vec{b}$.

These new iteration vectors where introduced in [Fea92b] for other purposes. A similar proposal, with a different numbering scheme has been made in [KP96].

2.3 Data Flow Analysis

2.3.1 Formal Solution

Suppose that we are given a program conforming to the restrictions of section 2.2.1. Let T be a statement in which an array M is read. Statement T will be called the *observation statement* in what follows. Let $\vec{b}$ be the iteration vector of T; the subscripts of M are affine functions of $\vec{b}$. In vector form, the reference to M may be written $\mathtt{M}[\vec{g}(\vec{b})]$.

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
1 :      v[j,k] := v[j,k]-v[i,k]*v[j,i]/v[i,i];
```

The iteration vector of $\mathtt{S}_1$ is $\langle 1, i, 1, j, 1, k, 1\rangle$. The indexing function, $\vec{g}$, is given by:

$$\vec{g}(\vec{b}) = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \vec{b}.$$

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

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

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

For each $\mathtt{S}_i$, there is a set of operations which write into $\mathtt{M}[\vec{g}(\vec{b})]$. Let $\mathtt{Q}_i(\vec{b})$ be this set. The set of all candidate sources is:

$$\mathtt{Q}(\vec{b}) = \bigcup_{i=1}^{n} \mathtt{Q}_i(\vec{b}).$$

Let us state the conditions which apply to a generic member, $\vec{a}$ of $\mathtt{Q}_i(\vec{b})$:

- **Existence Predicate**: $\vec{a}$ must be a legitimate iteration vector for $\mathtt{S}_i$:

$$E_{\mathtt{S}_i}\vec{a} \geq \vec{n}_{\mathtt{S}_i}. \tag{6}$$

- **Subscript Equations** : the subscripts of `M` must be the same at the read and write operations:
$$\vec{f_i}(\vec{a}) = \vec{g}(\vec{b}).$$
Note that this vector equation subsumes r scalar equations, where r is the *rank* of `M`. In writing this equation, we have taken into account the fact that the subscripts of `M` are guaranteed to be within `M` bounds.

- **Sequencing Predicate** $\vec{a}$ must be executed earlier than $\vec{b}$:
$$\vec{a} \ll \vec{b}.$$

- **Environment** : The observation statement must be executed:
$$E_{\mathtt{T}}\vec{b} \geq \vec{n}_{\mathtt{T}}.$$

From this we deduce the definition of Q_i:

$$\mathsf{Q}_i(\vec{b}) = \{\vec{a} \mid E_{\mathtt{S}_i}\vec{a} \geq \vec{n}_{\mathtt{S}_i}, \vec{a} \ll \vec{b}, \vec{f_i}(\vec{a}) = \vec{g}(\vec{b})\}. \tag{7}$$

The sets Q_i may still be subdivided according to the following observation. Under the restrictions of Sect. 2.2.1, the existence predicate and subscript equations generate a set of affine constraints. As we have seen earlier, the sequencing predicate is a disjunction of affine predicates $\prec_p$. Hence, Q_i is a union of polyhedra, or, rather, sets of integer points contained in polyhedra:

$$\mathsf{Q}_i^p(\vec{b}) \quad = \quad \{\vec{a} \mid E_{\mathtt{S}_i}\vec{a} \geq n_{\mathtt{S}_i}, \vec{a} \prec_p \vec{b}, \vec{f_i}(\vec{a}) = \vec{g}(\vec{b})\}, \tag{8}$$

$$\mathsf{Q}(\vec{b}) \quad = \quad \bigcup_{i=1}^{n} \bigcup_{p=0}^{N_{\mathtt{S}_i\mathtt{T}}} \mathsf{Q}_i^p(\vec{b}). \tag{9}$$

Finally, the source we are seeking is the lexicographic maximum of $\mathsf{Q}(\vec{b})$:

$$\varsigma(\vec{b}) = \max \bigcup_{i=1}^{n} \bigcup_{p=0}^{N_{\mathtt{S}_i\mathtt{T}}} \mathsf{Q}_i^p(\vec{b}). \tag{10}$$

In this paper, we will make repeated use of the following:

Property 1

$$\max \bigcup_{i=1}^{n} E_i = \max_{i=1}^{n}(\max E_i),$$

where the E_i are arbitrary subsets of a totally ordered set E, and where max *is the maximum operator associated to the order relation of E.*

The proof is trivial if none of the sets E_i is empty. If not, we have to introduce a special symbol, $\perp$, representing the undefined value, to stand in place of the maximum of an empty set. By convention, $\perp$ is less than any other value in any of the sets E_i:

$$\forall x \in E : \bot \ll x. \tag{11}$$

Application of the above property to (10) lead to the computation of

$$\begin{aligned} \varsigma_i^p(\vec{b}) &= \max \mathsf{Q}_i^p(\vec{b}) && (12) \\ \varsigma(\vec{b}) &= \max_{i=1}^{n} \max_{p=0}^{N_{\mathtt{S}_i,\mathtt{T}}} \varsigma_i^p(\vec{b}). && (13) \end{aligned}$$

The quantities $\varsigma_i^p(\vec{b})$ are known as *direct dependences* and were first defined by Brandes [Bra88].

To avoid multiple indices, we will renumber all possible candidates at all depths with a new index j. L will stand for the cardinal of the set of possible sources. (13) will be rewritten as :

$$\varsigma(\vec{b}) = \max\{\varsigma_j(\vec{b}) \mid j = 1, L\}. \tag{14}$$

Let us go back to the example in Figure 3. Consider the problem of finding the source of `c[i+j]` in statement $\mathtt{S}_2$. There are two candidates, $\mathtt{S}_1$ and $\mathtt{S}_2$ itself, and as a consequence, three functions ς_1^0, ς_2^0 and ς_2^1. The vector $\vec{b}$, in this case, has dimension 5: $\langle 2, i, 1, j, 1\rangle$. To simplify notations, only its variable components, i and j, will be taken into account.

Consider for instance the set $\mathsf{Q}_2(i,j)$. Its elements are five dimensional integer vectors $\langle 2, i', 1, j', 1\rangle$ 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. This implies that Q_2^1 is empty and $\varsigma_2^1 = \bot$.
- the limit constraints $0 \le i' \le n, 0 \le j' \le n$.

Examination of figure 4 shows that $\mathsf{Q}_2(i,j)$ is empty if $i = 0$ or $j = n$. If not empty, its lexical maximum is the vector $\langle 2, i-1, 1, j+1, 1\rangle$. This implies that to represent ς_2^0, we will need a conditional:

$$\varsigma_2^0(i,j) = \textbf{if } (i \ge 1 \wedge j < n) \textbf{ then } \langle 2, i-1, 1, j+1, 1\rangle \textbf{ else } \bot. \tag{15}$$

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

$$\varsigma_1^0 = \langle 1, i+j, 1\rangle.$$

Computing the lexicographic maximum of these values is now a straightforward exercise in algebra. The result is:

$$\varsigma(i,j) = \textbf{if } (i \ge 1 \wedge j < n) \textbf{ then } \langle 2, i-1, 1, j+1, 1\rangle \textbf{ else } \langle 1, i+j, 1\rangle. \tag{16}$$

To obtain this result, we have relied a lot on figure 4 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.

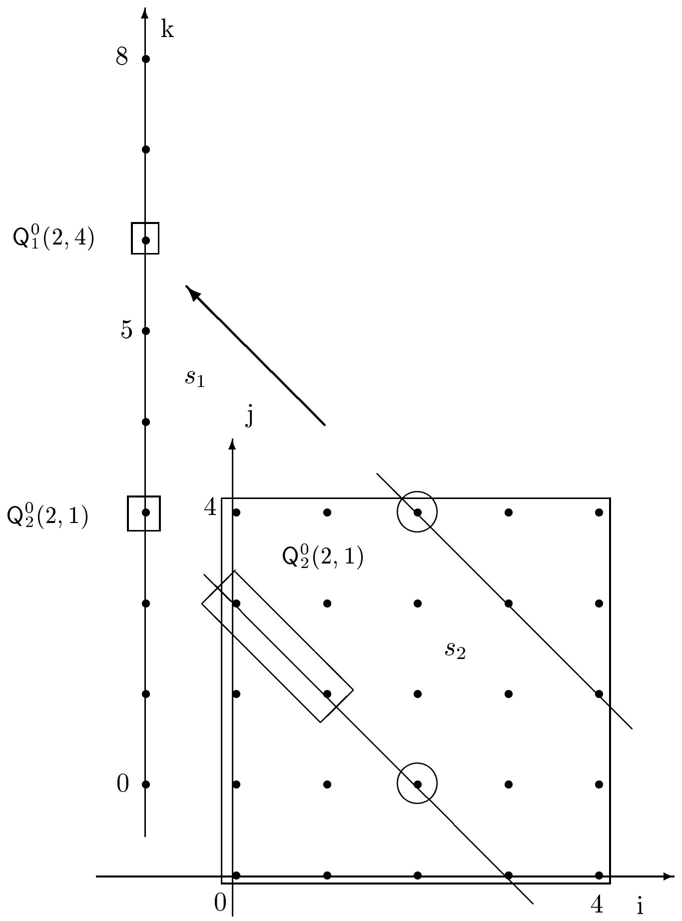

Figure 4: Computing the source function for the program of Figure 3

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.

2.3.2 Evaluation Techniques

Direct Dependences In this section, we will focus first on one particular direct dependence ς_i^p at a given depth p. When the original program conforms to the restrictions of section 2.2.1, all terms in formula (12) 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 `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 $\mathtt{S}_i <_{\text{text}} \mathtt{T}$ is false.

As a consequence, $\mathsf{Q}_i^p(\vec{b})$ is the set of integer vectors which lie inside a polyhedron. Finding its lexical maximum is a Parametric Integer Program (a PIP)[Fea88b]. A short description of an algorithm for solving PIP problems is given in the appendix. The parameters are the components of $\vec{b}$ and the structure parameters. Note that the components of $\vec{b}$ are not arbitrary; they must satisfy various constraints, among which is:

$$E_{\mathrm{T}}\vec{b} \geq \vec{n}_{\mathrm{T}}$$

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 Euclidean division by an integer. The solution is then expressed as a multistage conditional expression. The predicates are of the form $f(\vec{b}) \geq 0$, 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 result of this analysis is the direct dependence at depth p between the definition by $\mathtt{S}_i$ and the use in `T`. 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 $\mathtt{S}_i$.

Formula (15) 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.
      do i = 1, 2*n, 2
1       x(i) = 1
      end do
      do k = 1, 2*n
2       s = s + x(k)
      end do
```

The direct dependence from `x[2*i-1]` in $\mathtt{S}_1$ to `x[k]` in $\mathtt{S}_2$ is given by the following quast:

$$\varsigma_1^0(k) = \langle 1, \mathbf{if}\ 2((k+1) \div 2) - (k+1) \geq 0\ \mathbf{then}\ (k+1) \div 2\ \mathbf{else}\ \perp, 1\rangle.$$

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

Combining the direct dependences Consider now the problem of evaluating (14). This will be done in a sequential manner, by introducing:

$$\begin{aligned}\sigma_n &= \max\{\varsigma_j \mid j = 1, \ldots, n\},\\ \sigma_0 &= \perp.\end{aligned}$$

Obviously, $\varsigma(\vec{b}) = \sigma_L$ and we have the recurrence:

$$\sigma_n = \max(\sigma_{n-1}, \varsigma_n). \tag{17}$$

We are thus led to the evaluation of $\max(\sigma, \tau)$ where σ, τ are arbitrary quasts. We will use the term *extended quast* for any formula constructed from $\perp$ and quasi-linear vectors by the operations of selection (`if ... then ... else ...`) and taking a maximum.

Our problem is then to remove the maximum operator from an extended quast. This is done with the help of the following rules (and of their symmetrical counterparts, as the max operator is commutative).

Rule 1 $\max(\perp, \sigma) = \sigma$. *(This is simply a restatement of (11).)*

Rule 2 *If* $\sigma =$ **if** C **then** σ_1 **else** σ_2, *then:*

$$\max(\sigma, \tau) = \textbf{if } C \textbf{ then } \max(\sigma_1, \tau) \textbf{ else } \max(\sigma_2, \tau)$$

Rule 3 *If* u *and* v *are quasi linear vectors then*

$$\max(u, v) = \textbf{if } u \ll v \textbf{ then } v \textbf{ else } u.$$

The context of a node in a quast, C, is the conjunction of all the predicates which are asserted to be true as one follows the path from the root of the quast to the distinguished node. C is constructed by "anding" p if the leaf is in the true part of a conditional **if** p **then** ...., and by anding $\neg p$ if it is in the false part.

Rule 4 *Let* **if** p **then** σ **else** τ *be a subtree of a quast, and let* $\mathcal{C}$ *be its context. Then if* $\mathcal{C} \wedge p$ *is not feasible, replace the subtree by* τ. *Similarly, if* $\mathcal{C} \wedge \neg p$ *is not feasible, replace the subtree by* σ.

Rule 5 **if** C **then** σ **else** $\sigma = \sigma$.

Theorem 2 *If Rules 1 to 5 are oriented from left to right and used as rewrite rules, then their application to any extended quast always terminates.*

Proof Let us introduce the following metrics:

- The size of an extended quast, $|\sigma|$ is the number of nodes in the tree representation of σ. It is given by the following recursive definition:
  1. $|\perp| = |u| = 1$, where u is a quasi linear form.
  2. $|\textbf{if } p \textbf{ then } \sigma \textbf{ else } \tau| = 1 + |\sigma| + |\tau|$.
  3. $|\max(\sigma, \tau)| = 1 + |\sigma| + |\tau|$.

- The height of a max operator is simply the sum of the sizes of its arguments:
$$h(\max(\sigma,\tau)) = |\sigma| + |\tau|.$$
- $|\sigma|_{\mathbf{if}}$ is the number of **if** 's in an extended quast.

Rules 1 to 3 have the property that the max operator on the left has greater height than the (eventual) max operators on the right. In the case of 2, for instance, we have:

$$\begin{aligned}
h(\max(\mathbf{if}\ C\ \mathbf{then}\ \sigma_1\ \mathbf{else}\ \sigma_2,\tau)) &= 1 + |\sigma_1| + |\sigma_2| + |\tau|,\\
h(\max(\sigma_1,\tau)) &= |\sigma_1| + |\tau|,\\
h(\max(\sigma_2,\tau)) &= |\sigma_2| + |\tau|,
\end{aligned}$$

If there are further max operators inside τ, for instance, their height is left undisturbed by application of the rules. All other rules may only remove some max operators, without changing the height of those which are left undisturbed.

Finally, the effect of rules 4 and 5 is to remove some **if** operators. From these results we deduce that as the reduction of an extended quast proceeds, the maximum height of the max operators stays bounded by the maximum height in the original quast, H. Let us associate to each quast σ in the reduction a vector $\mu(\sigma)$ of dimension $H+1$, whose H first components are the histogram of "max" heights in reverse order, the last component being $|\sigma|_{\mathbf{if}}$. The first component of $\mu(\sigma)$ is the number of max's of maximum height, H. From the above discussion, we see that the effect of rules 1 to 3 is to decrease by one some component, i, of $\mu(\sigma)$. In the case of rule 2, two components of index $j,k > i$ are increased by one. Rules 4 and 5 may have the effect of decreasing some components of $\mu(\sigma)$ (if there are max operators in the discarded argument), and also to decreases by at least 1 the last component. The conclusion is that for all elementary reduction steps $\sigma \to \tau$, we have $\mu(\tau) \ll \mu(\sigma)$ in lexicographic order. Since lexicographic order on positive integer vectors is well founded, the reduction process must eventually stops, QED. Furthermore, as long as there is a max operator in the reduct, one of the rules 1 to 3 can be applied. Hence, when the reduction stops, there are no max operators in the result. ∎

In contrast to this result, it can be shown by counterexample that our rewriting system is not confluent, i.e. that the same extended quast can be reduced to several distinct quasts. However, since all rules are semantical equalities, it follows that all such reducts are semantically equal.

In the case of (16), we have to compute:

$$\sigma = \max(\perp, \max(\left\{\begin{array}{l}\mathbf{if}\ i \geq 1 \land j < n\\ \mathbf{then}\ \langle 2, i-1, 1, j+1, 1\rangle\\ \mathbf{else}\ \perp\end{array}\right., \langle 1, i+j, 1\rangle)).$$

We have successively:

$$\sigma = \max(\begin{cases} \textbf{if } i \geq 1 \wedge j < n \\ \textbf{then } \langle 2, i-1, 1, j+1, 1 \rangle \\ \textbf{else } \perp \end{cases}, \langle 1, i+j, 1 \rangle))$$

by rule 1, then

$$\sigma = \begin{cases} \textbf{if } i \geq 1 \wedge j < n \\ \textbf{then } \max(\langle 2, i-1, 1, j+1, 1 \rangle, \langle 1, i+j, 1 \rangle) \\ \textbf{else } \max(\perp, \langle 1, i+j, 1 \rangle) \end{cases}.$$

For the application of rule 3, we notice that $\langle 2, i-1, 1, j+1, 1 \rangle \ll \langle 1, i+j, 1 \rangle$ is always false. Use of this property is an example of rule 4. In the other arm of the conditional, rule 1 is applied again, giving the final result:

$$\sigma = \textbf{if } (i \geq 1 \wedge j < n) \textbf{ then } \langle 2, i-1, 1, j+1, 1 \rangle \textbf{ else } \langle 1, i+j, 1 \rangle.$$

2.4 Summary of the algorithm

Suppose that a compiler or another program processor has need to find the source of a reference to an array or scalar `M` in a statement `S`. The first step is to construct the candidate list, which comprises all statements `R` which modify `M` at all depths $0 \leq p \leq N_{\mathtt{RS}}$. If a standard dependence analysis is available, this list can be shortened by eliminating empty candidate sets, which correspond to non existent dependences.

The ordering of the candidate set is a very important factor for the complexity of the method. Experience has shown that the best compromise is to list the candidates in order of decreasing depth. For equal depth candidates, it is best to follow the textual order backward, starting from the distinguished reference, up to the beginning of the program, and then to loop back to the end of the program.

Similarly, if rule 4 is used too sparingly, the resulting quasts will have many dead branches, thus increasing the complexity of the final result. Conversely, if used too often, it will result in many unsuccessful attempts at simplification, also increasing the complexity. A good compromise is the following:

- When computing a step of the recurrence (17) we will always suppose that rule 4 has been applied exhaustively to σ_{n-1}.

- In the evaluation of σ_n, rule 2 should be applied by priority on the left argument. As long as reductions are still possible on σ_{n-1}, there is no need to apply rule 4. All contexts that can be constructed here are feasible, because they either come from σ_{n-1} or ς_n. The first quast has been simplified in the previous step, and the second one comes from PIP, which does not generate dead branches.

- As soon as the application of rules 2 or 3 to ς_n starts, simplification by rule 4 should be attempted.

As a last remark, one can show (see Sect. 3.3.3 of [Fea91]) that the complete knowledge of the iteration vector is not needed when applying the max operator to sources of differing depths. In this way, one can predict beforehand whether a direct dependence can have influence on the source or not, and avoid computing it in the latter case.

If these rules are followed, the results of array dataflow analysis are surprisingly simple. A limited statistical analysis in [Fea91] shows that the mean number of leaves per source is about two. The probable reason is that good programmers do a kind of dataflow analysis "in their head" to convince themselves that their program is correct. If the result is too complicated, they decide that the program is not well written and start again.

2.5 Related Work

Another approach to Array Dataflow Analysis has been proposed by Pugh and his associates (see e.g. [PW93, Won95]). The approach consists in reverting to the basic definition of the maximum of a set. u is the maximum of a totally ordered set Q iff:

$$u \in Q \wedge \neg \exists v \in Q : u \prec v.$$

Let us consider the definition (12) of a set of candidate sources. According to the above definition, its maximum, $\varsigma_i(\vec{b})$ is defined by:

$$\varsigma_i(\vec{b}) \in \mathsf{Q}_i(\vec{b}) \wedge \neg \exists \vec{c} : \varsigma_i(\vec{b}) \prec \vec{c} \prec \vec{b} \wedge \vec{c} \in \mathsf{Q}_i(\vec{b}). \tag{18}$$

In words, $\varsigma_i(\vec{b})$ is the direct dependence from $\mathtt{S}_i$ to $\vec{b}$ iff $\varsigma_i(\vec{b})$ is in flow dependence to $\vec{b}$ and if there is no other operation in flow dependence to $\vec{b}$ which is executed between $\varsigma_i(\vec{b})$ and $\vec{b}$.

The formula (18) is written in a subset of Presburger logic (that part of first order logic which deals with the theory of addition of positive numbers), which is known to be decidable. Pugh has devised an algorithm, the Omega test [Pug91] which is able to simplify formulas such as (18). The result is a relation between the source $\varsigma_i(\vec{b})$ and $\vec{b}$. It has been checked that this relation is equivalent to the quast which is found by our method.

Some authors [MAL93, HT94] have devised fast methods for handling particular cases of the source computation. The idea is to solve the set of equations in the definition of $\mathsf{Q}_i^p(\vec{b})$ by any integer linear solver (e.g. by constructing the Hermite normal form of the equation matrix). Suppose that this system uniquely determines $\vec{a}$ as a function of $\vec{b}$: $\vec{a} = f(\vec{b})$. It remains only to substitute $f(\vec{b})$ for $\vec{a}$ in the inequalities. The result is the existence condition for the solution, which is $f(\vec{b})$ if this condition is satisfied, and $\perp$ if not. One must revert to the general algorithm if there are not enough equations to fix the value of the maximum.

3 Approximate Array Dataflow Analysis

To go beyond the static control model, one has to handle `while` loops, arbitrary subscripts and tests, and, most important, modular programming (subroutines and function calls). Let us first introduce the following convention. Constructs occurring in the control statement of a program (`do` loop bounds, `while` loops and tests predicates, subscripts) will be classified as *tractable* and *intractable* according to their complexity. Affine constructs are always tractable, while the definition of intractable constructs is somewhat subjective and may depends on the analysis tools which are available at any given time. Tractable predicates in tests can always be replaced by restrictions on the iteration domain of the surrounding loop nest. Similarly, a tractable predicate in a `while` loop indicates that the loop can be transformed into a `do` loop. We will suppose that such simplifications have been applied before the approximate analysis starts.

In this section we will be interested in `while` loops and tests. Non linear subscripts can be handled in the same framework, but they need rather complicated notations. The reader is referred to [BCF97] for this extension.

As a matter of convenience, we will suppose here that `while` loops have an explicit loop counter, according to the PL/I convention:

```
do c = 1 while p(...)
```

The `while` loop counter may even be used in array subscripts.

When constructing iteration vectors, tests branches are to be considered as new nodes being numbered 1 and 2. In accordance with our conventions for static control programs, these nodes always are compound statements, whatever the number of their components. For instance, in example E3 below, the iteration vector of Statement $\mathtt{S}_2$ is $\langle 1, x, 1, 2, 1\rangle$. The first 1 is the index of the `do` loop in the whole program, and the second one is the index of the test in the `do` loop body. The 2 indicates that the subject statement is in the false part of the test.

With these conventions, we can transpose to this new program model most of the notations we introduced for static control programs. Iteration vectors may include `while` loop counters and the definition of the sequencing predicate does not change.

3.1 From ADA to FADA

As soon as we extend our program model to include conditionals, `while` loops, and `do` loops with intractable bounds, the set Q_i^p of (8) is no longer tractable at compile time. The reason is that condition (6) may contain intractable terms. One possibility is to ignore them. In this way, (6) is replaced by:

$$E'_{\mathtt{S}_i}\vec{a} \geq \vec{n}'_{\mathtt{S}_i}, \tag{19}$$

where E' and $\vec{n}'$ are similar to E and $\vec{n}$ in (6) with the intractable parts omitted. We may obtain approximate sets of candidate sources:

$$\widehat{\mathsf{Q}}_i^p(\vec{b}) = \{\vec{a} \mid E'_{\mathtt{S}_i}\vec{a} \geq \vec{n}'_{\mathtt{S}_i}, \vec{a} \prec_p \vec{b}, \vec{f}_i(\vec{a}) = \vec{g}(\vec{b})\}. \tag{20}$$

However, we can no longer say that the direct dependence is given by the lexicographic maximum of this set, since the result may precisely be one of the candidates which is excluded by the nonlinear part of of the iteration domain of $\mathtt{S}$. One solution is to take all of $\widehat{\mathsf{Q}}_i(\vec{b})$ as an approximation to the direct dependence. If we do that, and with the exception of very special cases, computing the maximum of approximate direct dependences has no meaning, and the best we can do is to use their union as an approximation. Can we do better than that? Let us consider some examples.

```
    program E1
    do x = 1 while ...
1     s = ...
    end do
```

```
2    s = ...
3    ... = ... s ...
     end
```

What is the source of `s` in Statement $\mathtt{S}_3$? There are two possibilities, Statements $\mathtt{S}_1$ and $\mathtt{S}_2$. In the case of $\mathtt{S}_2$, everything is linear, and the source is exactly $\langle 2 \rangle$. Things are more complicated for $\mathtt{S}_1$, since we have no idea of the iteration count of the `while` loop. We may, however, give a name to this count, say N, and write the set of candidates as:

$$\mathsf{Q}_1^0 = \{\langle 1, x, 1 \rangle \mid 1 \leq x \leq N\}.$$

We may then compute the maximum of this set, which is

$$\varsigma_1^0 = \textbf{if } N > 0 \textbf{ then } \langle 1, N, 1 \rangle \textbf{ else } \perp.$$

The last step is to take the lexicographic maximum of this result and $\langle 2 \rangle$, which is simply $\langle 2 \rangle$. This is much more precise than the union of all possible sources. The trick here has been to give a name to an unknown quantity, N, and to solve the problem with N as a parameter. It so happens here that N disappears in the solution, giving an exact result.

Consider now:

```
     program E2
     do x = 1 while ...
1      s(x) = ...
     end do
     do k = 1,n
2      ... = ... s(k) ...
     end do
     end
```

With the same notations as above, the set of candidates for the source of `s(k)` in $\mathtt{S}_3$ is:

$$\mathsf{Q}_1^0(k) = \{\langle 1, x, 1 \rangle \mid 1 \leq x \leq N, x = k\}.$$

The direct dependence is to be computed in the environment $1 \leq k \leq n$ which gives: $\textbf{if } k \leq N \textbf{ then } \langle 1, k, 1 \rangle \textbf{ else } \perp$. Here, the unknown parameter N has not disappeared. The best we can do is to say that we have a source *set*, or a *fuzzy* source, which is obtained by taking the union of the two arms of the conditional:

$$\varsigma(k) \in \{\langle 1, k, 1 \rangle, \perp\}.$$

Equivalently, by introducing a new notation $\Sigma(\vec{b})$ for the source set at iteration $\vec{b}$, this can be written:

$$\Sigma(k) = \{\langle 1, k, 1 \rangle, \perp\}.$$

The fact that in the presence of of intractable constructs, the results are no longer sources but sets of possible sources justifies the name Fuzzy ADA which has been given to the method. FADA gives exact results (and reverts to ADA) when the source sets are singletons.

Our last example is slightly more complicated: we assume that $n \geq 1$,

```
      program E3;
      begin
1 :     for x := 1 to n do
2 :     begin
3 :       if ... then
4 :       begin
5 :         s := ...
          end
          else
6 :       begin
7 :         s := ...
          end
        end;
8 :     ... := s ....
       end
```

What is the source of s in Statement S_8? We may build an approximate candidate set from S_5 and another one from S_7. Since both are approximate, we cannot do anything beside taking their union, and the result is highly inaccurate.

Another possibility is to partition the set of candidates according to the value x of the loop counter. Let us introduce a new Boolean function $b(x)$ which represents the outcome of the test at iteration x. The x-th candidate may be written[3]:

$$\tau(x) = \textbf{if } b(x) \textbf{ then } \langle 1, x, 1, 1, 1\rangle \textbf{ else } \langle 1, x, 1, 2, 1\rangle.$$

We then have to compute the maximum of all these candidates (this is an application of Property 1). It is an easy matter to prove that:

$$x < x' \Rightarrow \tau(x) \prec \tau(x').$$

Hence the source is $\tau(n)$. Since we have no idea of the value of $b(n)$, we are lead again to the introduction of a fuzzy source:

$$\Sigma = \{\langle 1, 1, n, 1, 1\rangle, \langle 1, 1, n, 2, 1\rangle\}. \tag{21}$$

Here again, notice the far greater precision we have been able to achieve. However, the technique we have used here is not easily generalized. Another way of obtaining the same result is the following. Let $\mathbf{L} = \{x \mid 1 \leq x \leq n\}$. Observe that the candidate set from S_1 (resp. S_2) can be written $\{\langle\{1, x, 1, 1, 1\rangle \mid x \in \mathbf{D}_1 \cap \mathbf{L}\}$ (resp. $\{\langle 1, x, 1, 2, 1\rangle \mid x \in \mathbf{D}_2 \cap \mathbf{L}\}$) where

$$\mathbf{D}_1 = \{x \mid b(x) = \textbf{ true } \} \text{ and } \mathbf{D}_2 = \{x \mid b(x) = \textbf{ false}\}.$$

Obviously,

$$\mathbf{D}_1 \cap \mathbf{D}_2 = \emptyset, \tag{22}$$

and

$$\mathbf{D}_1 \cup \mathbf{D}_2 = \mathbb{Z}. \tag{23}$$

[3] Observe that the ordinals in the following formula do not correspond to the statement labels in the source program. These labels have been introduced for later use (see Sect. 3.3).

We have to compute

$$\beta = \max(\max \mathbf{D}_1 \cap \mathbf{L}, \max \mathbf{D}_2 \cap \mathbf{L}).$$

Using property 1 in reverse, (23) implies:

$$\beta = \max \mathbf{L}. \tag{24}$$

By (22) we know that β belongs either to $\mathbf{D}_1$ or $\mathbf{D}_2$ which gives again the result (21).

To summarize these observations, our method will be to give new names (or *parameters*) to the result of maxima calculations in the presence of nonlinear terms. These parameters are not arbitrary. The sets they belong to – the parameters domains – are in relation to each others, as for instance (22-23). These relations can be found simply by examination of the syntactic structure of the program, or by more sophisticated techniques. From these relations between the parameter domains follow relations between the parameters, like (24), which can then be used to simplify the resulting fuzzy sources. In some cases, these relations may be so precise as to reduce the fuzzy source to a singleton, thus giving an exact result.

3.2 Introducing Parameters

In the general case, any statement in the program is surrounded by tests and loops, some of which are tractable and some are not. Tractable tests and loops give the linear part of the existence predicate, definition (19) above. To the non tractable parts we may associate a set d_i such that operation $\vec{a}$ exists iff:

$$E'_{\mathtt{S}_i}\vec{a} \geq \vec{n}'_{\mathtt{S}_i} \wedge \vec{a} \in d_i. \tag{25}$$

The observation which allows us to increase the precision of FADA is that in many cases d_i has the following property:

$$\vec{a}[1..p_i] = \vec{b}[1..p_i] \Rightarrow (\vec{a} \in d_i \equiv \vec{b} \in d_i) \tag{26}$$

for a p_i which is less than the depth of $\mathtt{S}_i$. This is due to the fact that loops and tests predicates cannot take into account variables which are not defined at the point they are evaluated, as is the case for inner loop counters. Usually, p_i is the number of (`while` and `do`) loops surrounding the innermost non tractable construction around $\mathtt{S}_i$. This depth may be less than this number, in case the intractable predicate does not depend on some variables, but this can be recognized only by a semantics analysis which is beyond the scope of this paper.

A *cylinder* is a set C of integer vectors such that there exists an integer p – the depth – with the property:

$$\vec{a} \in C \wedge \vec{a}[1..p] = \vec{b}[1..p] \Rightarrow \vec{b} \in C. \tag{27}$$

The depth of cylinder C will be written $\delta(C)$.

The above discussion shows that to each d_i we may associate a cylinder C_i by the definition:

$$\vec{a} \in C_i \equiv \exists \vec{b} \in d_i : \vec{a}[1..p_i] = \vec{b}[1..p_i],$$

with the property:

$$E'_{\mathsf{S}_i}\vec{a} \geq \vec{n}'_{\mathsf{S}_i} \wedge \vec{a} \in d_i \equiv E'_{\mathsf{S}_i}\vec{a} \geq \vec{n}'_{\mathsf{S}_i} \wedge \vec{a} \in C_i.$$

The depth of C_i is bounded upward by the number of loops surrounding $\mathtt{S}_i$; a more precise analysis may show that it has a lower value.

With these convention, the set of candidate sources becomes:

$$\mathsf{Q}_i^p(\vec{b}) = \{\vec{a} \mid E'_{\mathsf{S}_i}\vec{a} \geq n'_{\mathsf{S}_i}, \vec{a} \in C_i, \vec{a} \prec_p \vec{b}, \vec{f}_i(\vec{a}) = \vec{g}(\vec{b})\}. \tag{28}$$

Let us introduce the following sets:

$$\widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\alpha}) = \{\vec{a} \mid E'_{\mathsf{S}_i}\vec{a} \geq n'_{\mathsf{S}_i}, \vec{a}[1..p_i] = \vec{\alpha}, \vec{a} \prec_p \vec{b}, \vec{f}_i(\vec{a}) = \vec{g}(\vec{b})\}. \tag{29}$$

$\widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\alpha})$ is the intersection of $\widehat{\mathsf{Q}}_i^p(\vec{b})$ with the hyperplane $\vec{a}[1..p_i] = \vec{\alpha}$. (28) can be rewritten:

$$\mathsf{Q}_i^p(\vec{b}) = \bigcup_{\alpha \in C_i} \widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\alpha}). \tag{30}$$

Another use of property 1 gives:

$$\varsigma_i^p(\vec{b}) = \max \mathsf{Q}_i^p(\vec{b}) = \max_{\vec{\alpha} \in C_i} (\max \widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\alpha})) \tag{31}$$

Now $\widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\alpha})$ is a polyhedron, as is evident from (29). Hence its lexicographic maximum,

$$\widehat{\varsigma}_i^p(\vec{b}, \vec{\alpha}) = \max \widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\alpha}) \tag{32}$$

can be computed by just another application of PIP. In fact, the presence of the additional inequalities $\vec{a}[1..p_i] = \vec{\alpha}$ may simplify this calculation. We then have:

$$\varsigma_i^p(\vec{b}) = \max_{\vec{\alpha} \in D_i} \widehat{\varsigma}_i^p(\vec{b}, \vec{\alpha}). \tag{33}$$

The maximum in the above formula is reached at some point of C_i. This point is a function of i, p and $\vec{b}$, written as $\vec{\beta}_i^p(\vec{b})$ and is known as one of the *parameters of the maximum* of the program. The direct dependence is now given by:

$$\varsigma_i^p(\vec{b}) = \widehat{\varsigma}_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b})). \tag{34}$$

At this point, we can go on as we did in the case of exact analysis:

- Compute all parametric direct dependences by (32).
- Combine the direct dependences by rules 1 to 5.
- In the end result, quantify over all possible values of the parameters, so as to get source sets.

This procedure does not give precise results, since we lose all information about relations between parameters of the maximum. Our aim now is to explain how to find these relations and how to use them to narrow the final result.

3.3 Taking Properties of Parameters into Account

The sets C_i, C_j for $i \neq j$ may be interrelated, depending on the position of statements $\mathtt{S}_i$, $\mathtt{S}_j$ in the abstract syntax tree. An example of this situation has been observed for statements $\mathtt{S}_5$ and $\mathtt{S}_7$ of program E3. These relations induce relations between the corresponding parameters, which have to be taken into account when combining direct dependences. The relations on the C_i sets may have several origins. The most obvious ones are associated to the structure of the source program, as in the case of E3. It may be that other relations are valid, due for instance to the equality of two expressions. Here again, this situation can be detected only by semantics analysis and is outside the scope of this paper.

The structural relations among the C_i can be found by the following algorithm:

- The outermost construction of the source program (by our convention, a compound statement), is associated to the unique zero-depth cylinder, which includes all integer vectors of any length, and can be written as $\mathbb{Z}^*$.
- If C_0 is associated to:

  ```
  begin S1; .... Sn end
  ```

  then $C_i = C_0$.
- If C_0 is associated to:

  ```
  if p then S1 else S2
  ```

  where p is intractable, then the cylinders associated to $\mathtt{S}_1$, C_1 and $\mathtt{S}_2$, C_2 have the same depth as C_0 and are such that:

  $$C_1 \cap C_2 = \emptyset,\ C_1 \cup C_2 = C_0.$$

  If p is tractable, $C_0 = C_1 = C_2$.
- If C_0 is associated to a `for`:

  ```
  for ..... do S1
  ```

  or to a `while`:

  ```
  do ... while ... S1
  ```

  and if these loops are intractable, then the cylinder C_1 associated to $\mathtt{S}_1$ has depth $\delta(C_0) + 1$ and is such that:

  $$C_1 \subseteq C_0.$$

  Otherwise, $C_1 = C_0$.

The relation between ADA and FADA In the case where all enclosing statements of an assignment $\mathtt{S}_i$ are tractable, it is easy to prove that $C_i = \mathbb{Z}^*$. The condition $\vec{a} \in C_i$ is trivially satisfied in (28). Hence, in that case, FADA defaults to ADA. Provided this case is detected soon enough, one and the same algorithm can be used for all programs, and the precision of the results will depends on the presence or absence of intractable control constructs.

Characterization of the Parameters of the Maximum The main observation is that each parameters is itself a maximum. Note first that from (29) follows:

$$\vec{\beta}_i^p(\vec{b}) = \varsigma_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b}))[1..p_i].$$

Suppose now that Q is an arbitrary set of vectors all of which have dimension at least equal to p. Let us set:

$$Q\mid_p = \{x[1..p] \mid x \in Q\}.$$

The properties of the lexicographic order insure that:

$$(\max Q)[1..p] = \max Q\mid_p .$$

In our case, this gives:

$$\begin{aligned}
\vec{\beta}_i^p(\vec{b}) &= \varsigma_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b}))[1..p_i] \\
&= (\max \widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b})))[1..p_i] \\
&= \max \widehat{\mathsf{Q}}_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b}))\mid_{p_i} \\
&= \max(C_i \cap \widehat{\mathsf{Q}}_i^p(\vec{b})\mid_{p_i}) \qquad (35)
\end{aligned}$$

where $\widehat{\mathsf{Q}}_i^p(\vec{b})$ is the "polyhedral envelope" of all possible sources at depth p (see (20)). This formula fully characterizes the parameters of the maximum and will be used repeatedly to obtain relations between them.

Another set of relations is given by depth considerations. Note that from (29) follows:

$$\varsigma_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b}))[1..p_i] = \vec{\beta}_i^p(\vec{b}),$$

and

$$\varsigma_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b}))[1..p] = \vec{b}[1..p],$$

provided the set $\varsigma_i^p(\vec{b}, \vec{\beta}_i^p(\vec{b}))$ is not empty. Now, in (35), we can exclude the case where $\widehat{\mathsf{Q}}_i^p(\vec{b})$ is empty, since this can be decided *a priori* by integer linear programming. If such is the case, statement $\mathtt{S}_i$ is simply excluded from the list of candidates at depth p. Hence, either C_i is empty, in which case, by (35), we set $\vec{\beta}_i^p(\vec{b}) = \perp$, or else the above relations apply. Let us set $m_i = \min(p, p_i)$. We obtain:

$$\vec{\beta}_i^p(\vec{b})[1..m_i] = \vec{b}[1..m_i] \vee \vec{\beta}_i^p(\vec{b}) = \perp. \qquad (36)$$

Exact Cases of FADA Among the C_i there is the set corresponding to the observation statement, C_ω. Since our convention is that the observation statement is executed, we have $\vec{b} \in C_\omega$, hence C_ω is not empty. It may happen that the results of structural analysis imply that $C_i = C_\omega$. Suppose that $p \geq p_i = p_\omega$. From (36) we deduce:

$$\vec{\beta}_i^p = \vec{b}[1..p_i].$$

This allows us to remove the nonlinear condition $\vec{a} \in C_i$ from (28) before computing its maximum.

In the case where the innermost intractable statement is a `while` or a `do` loop, we can go a step further since C_i now has the property:

$$x \in C_i \wedge (\vec{a}[1..p_i - 1] = \vec{b}[1..p_i - 1] \wedge \vec{a}[p_i] < \vec{b}[p_i]) \Rightarrow \vec{b} \in C_i.$$

This means that the exactness condition is in that case:

$$p \geq p_i - 1.$$

This enable us to solve exactly such problems as the source of `s` in:

```
    do c = 1 while ...
1     s = s + ....
    end do
```

Here the candidate and observation statements are both $\mathtt{S}_1$. $p_1 = 1$ and $p = 0$. The exactness condition is satisfied, and the source is:

$$\begin{aligned} \varsigma_1^0(c) &= \max\{\langle 1, c', 1\rangle \mid 1 \leq c', c' < c\} \\ &= \textbf{if } c > 1 \textbf{ then } \langle 1, c-1, 1\rangle \textbf{ else } \bot. \end{aligned}$$

From Parameter Domains to Parameters of the Maximum It remains to study the case where the structural analysis algorithm has given non trivial relations between parameters domains. The associated relations between parameters can be deduced from (35) by Prop. 1 and the following trivial properties:

Property 3 *If $C \cap D = \emptyset$, then:*

$$(\max C = \bot \wedge \max D = \bot) \vee \max C \neq \max D.$$

Property 4 *If $C \subseteq D$ then:*

$$\max C \underline{\ll} \max D.$$

As a consequence, since $C \cap D \subseteq C$, we have

$$\max(C \cap D) \underline{\ll} \max C, \tag{37}$$

and the symmetrical relation.

Example E3 **revisited** The observation statement is $\mathtt{S}_8$. It is enclosed in no loops. Hence, the $\vec{b}$ vector is of zero length, and will be omitted in the sequel. There are two candidate sources, $\mathtt{S}_5$ and $\mathtt{S}_7$, whose iteration vectors are of length one and will be denoted as x. It that case, lexicographic order defaults to the standard order among integers.

The parametric sources are:

$$\widehat{\varsigma}_5^0(\alpha) = \max\{x \mid 1 \leq x \leq n, x = \alpha\} = \textbf{if } 1 \leq \alpha \leq n \textbf{ then } \langle 1, \alpha, 1, 1, 1 \rangle \textbf{ else } \bot.$$

$$\widehat{\varsigma}_7^0(\alpha) = \max\{x \mid 1 \leq x \leq n, x = \alpha\} = \textbf{if } 1 \leq \alpha \leq n \textbf{ then } \langle 1, \alpha, 1, 2, 1 \rangle \textbf{ else } \bot.$$

The structural analysis algorithm gives the following relations:

$$\begin{array}{ccc} C_1 = C_0 & , & C_8 = C_0 \\ C_2 = C_1 & , & C_3 = C_2 \\ C_4 \cup C_6 & = & C_3 \\ C_4 \cap C_6 & = & \emptyset \\ C_5 = C_4 & , & C_7 = C_6. \end{array}$$

Here, only C_5 and C_7 are interesting. Remembering that $C_0 = \mathbb{Z}^*$, all other sets can be eliminated, giving:

$$C_5 \cup C_7 = \mathbb{Z}^*, \ C_5 \cap C_7 = \emptyset.$$

The depths p_5 and p_7 are both equal to 1. From Equ. (35) we deduce:

$$\vec{\beta}_5^0 = \max(C_5 \cap \widehat{\mathbf{Q}}_5^0) = \max(C_5 \cap [1, n]).$$

Similarly,

$$\vec{\beta}_7^0 = \max(C_5 \cap [1, n]).$$

From the above relations, we deduce:

$$(C_5 \cap [1, n]) \cap (C_7 \cap [1, n]) = \emptyset$$

and

$$(C_5 \cap [1, n]) \cup (C_7 \cap [1, n]) = \mathbb{Z}^* \cap [1, n] = [1, n].$$

This equality can be interpreted as two inclusions from left to right, giving by Prop. 4:

$$\vec{\beta}_5^0 \leq n, \ \vec{\beta}_7^0 \leq n,$$

or as an inclusion from right to left, giving:

$$n \leq \max(\vec{\beta}_5^0, \vec{\beta}_7^0).$$

Lastly, we deduce from the first relation that:

$$(\vec{\beta}_5^0 = \vec{\beta}_7^0 = \bot) \vee \vec{\beta}_5^0 \neq \vec{\beta}_7^0.$$

Suppose now that the maximum of $\vec{\beta}_5^0$ and $\vec{\beta}_7^0$ is $\vec{\beta}_5^0$. It is easily seen that this implies:

$$\vec{\beta}_5^0 = n, \vec{\beta}_7^0 < n.$$

In the reverse situation, the conclusion is:

$$\vec{\beta}_7^0 = n, \vec{\beta}_5^0 < n.$$

Hence, the final source is given by:

$$\varsigma = \begin{cases} \textbf{if } \vec{\beta}_5^0 = n \wedge \vec{\beta}_7^0 < n \\ \textbf{then } \max(\varsigma_5^0(\vec{\beta}_5^0), \varsigma_5^0(\vec{\beta}_5^0))/\{\vec{\beta}_5^0 = n, \vec{\beta}_7^0 < n\} \\ \textbf{else } \max(\varsigma_5^0(\vec{\beta}_5^0), \varsigma_5^0(\vec{\beta}_5^0))/\{\vec{\beta}_7^0 = n, \vec{\beta}_5^0 < n\} \end{cases}$$

where the notation $q/\{C\}$ indicates that the quast q is to be evaluated by rules 1 to 5 in the context C. We leave it to the reader to verify that the result is:

$$\varsigma = \textbf{if } \vec{\beta}_5^0 = n \wedge \vec{\beta}_7^0 < n \textbf{ then } \langle 1, n, 1, 1, 1\rangle \textbf{ else } \langle 1, n, 1, 2, 1\rangle.$$

3.4 Eliminating Parameters

The result of the above computation can be considered as a parametric representation of the fuzzy source: as the parameters take all possible values, the result visits all possible sources. In some cases, this is exactly what is needed for further analysis. In most case, however, more compact representations are enough. This can be obtained by the following process.

Let $\sigma(\vec{\beta})$ be a leaf of the fuzzy source, where $\vec{\beta}$ symbolizes all parameters occurring in the leaf. Parameter elimination uses the two rules:

Rule 6 *A leaf $\sigma(\vec{\beta})$ in context C is replaced by the set:*

$$\{\sigma(\vec{\beta}) \mid \vec{\beta} \in C\}.$$

Note that after application of this rule, the variables of $\vec{\beta}$ become bound variables and do no longer occur in the result.

Rule 7 *A conditional* **if** $p(\vec{\beta})$ **then** A **else** B *where A and B are sets which do not depend on $\vec{\beta}$ is replaced by $A \cup B$.*

Application of these rules to the result of the analysis of `E3` gives the fuzzy source:

$$\Sigma = \{\langle 1, n, 1, 1, 1\rangle, \langle 1, n, 1, 2, 1\rangle\}.$$

Observe that rules 6 and 7 are consistent with rule 4. If the context of a leaf is unfeasible, the leaf can be removed by rule 4. It can also be transformed into the empty set by rule 6, and it will then disappear at the next application of rule 7.

3.5 Related Work

3.5.1 Pugh and Wonnacott's Method

Pugh and Wonnacott [Won95] have extended the Omega calculator for handling uninterpreted functions in logical formulas. This allows them to formulate problems of dataflow analysis in the presence of intractable constructs. They simply introduce a function to represent the value of the construct as a function of the surrounding loop counters. These functions may be used to represent the number of iteration of a `while` loop (see N in the analysis of example `E1` in Sect. 3.1) or the outcome of a test (see b for example `E3` in the same section). When we say that a construct has depth p_i, it means that the corresponding function has as arguments the p_i outermost loop counters.

The problem with this approach is that adding one uninterpreted function to Presburger logic renders it undecidable. Hence, Pugh and Wonnacott have to enforce restrictions to stays within the limits of decidability. They have chosen to partition the variables in a logical formula into input and output variables, and to use only uninterpreted functions which depends either on the input or output variables but not both. Applying a function to anything else (e.g. a bound variable inside an inner quantifier) is forbidden and is replaced by the uninterpreted symbol *unknown*. This restriction is rather *ad hoc* and it is difficult to assert its effect on the power of Pugh and Wonnacott's system. In fact, we know of several examples which they cannot handle but which can be solved by FADA: `E3` is a case in point. In the case of FADA, D. Barthou et. al. have proved in [BCF97] that their system of relations between parameters of the maximum are correct and complete, i.e. that no potential source is missed, and that each element of a source set can be a source for some realization of the intractable predicates.

On the other hand, Pugh and Wonnacott have included some semantical knowledge in their system. When assigning functions to intractable constructs, they identify cases in which two constructs are equal and assign them the same function. This is easily done by first converting the source program in Static Single Assignment (SSA) form. In SSA form, syntactically identical expressions are semantically equal. The detection of equal expression is limited to one basic bloc. This method allows them to handle examples such as:

```
      program E4;
      begin
        for i := 1 to n do
        begin
          if p(i) >= 0 then
1 :         s := ...;
          if p(i) < 0 then
2 :         s := ...;
        end;
        ... := s ...;
      end
```

in which the key to the solution is recognizing that `p(i)` has the same value in the two tests. We could have introduced a similar device in FADA; the result of the analysis could have been translated in term of the C_i sets (here, we would have got the same relations as in the case of `E3`) and the analysis would have then proceeded as above. We have chosen to handle first the semantical part of FADA. Recognizing equal and related expressions is left for future work, and we intend to do it with more powerful devices than SSA conversion (see [BCF97]).

3.5.2 Abstract Interpretation

As is well known, in denotational semantics, the aim is to build the input/output function of a program, which gives the final state of the computer memory in term of its initial state. This function is built in term of simpler functions, which give the effect of each statement and the value of each expression. These functions in turn are obtained by applying compilation functions to abstractions of the program text. The definitions of the compilation functions are constructive enough to enable a suitable interpreter to execute them. As many researchers have observed, these function definitions are quite similar to ML programs.

The basic idea of abstract interpretation [CC77] is to define other, non standard semantical functions. Obviously, this is interesting only if a nonstandard semantics can be considered in some sense as an approximation of standard semantics. This is formalized using the concept of Galois connection between the domains of the abstract and standard semantics.

An example of the use of these ideas for analysis of array accesses is found in [CI96]. In this work, the results of the analysis are *regions*, i.e. subsets of arrays as defined by constraints on their subscripts [TIF86]. Several types of regions are defined. For instance, the WRITE region of a statement is the set of array cells which may be modified when the statement is executed. The IN region is the set of array cells whose contents are used in a calculation.

When designing such an analysis, one has to select a finite representation for regions. In the quoted work, regions are convex polyhedra in the subscript space. Less precise representations have been suggested, see for instance [GLL95] for the concept of regular sections. In the same way as the standard semantics has operators to act on arrays, the nonstandard semantics must have operators to act on regions. These operators are intersection, union, subtraction and projection (which is used for computing the effect of loops). Depending on the representation chosen, these operators may be closed or not. For instance, intersections of polyhedra are polyhedra, but unions are not. In case of unclosed operators, one has to defined a closed approximation: for the union of polyhedra, one takes usually the convex hull of the real union.

One sees that there are two sources of approximation in region analysis. One comes from the choice of region representation. For instance, convex polyhedra are more precise than regular sections, but are not precise enough to represent frequently occurring patterns, like:

```
do i = 1,n
  m(2*i-1) = 0.
end do
```

The corresponding write region, in [CI96] notation, is $\langle \mathtt{m}(\phi), 1 \leq \phi \leq 2n-1\rangle$, which is only an approximation of the exact region, $\langle \mathtt{a}(\phi), \phi = 2\psi - 1, 1 \leq \psi \leq n\rangle$.

The second source of approximation is the same as the one in FADA: the source program may contain intractable constructs. Approximate regions are constructed by ignoring the intractable terms, in the spirit of (20).

ADA and FADA represent their results not as convex polyhedra but as finite unions of Z-polyhedra (the intersection of a polyhedron and a Z-module, see the appendix). This representation is inherently more precise and has enough power to represent exactly all regions occurring in the analysis of static control programs. An interesting open problem is the following: is it possible to reformulate the method of [CI96] in term of unions of Z-polyhedra, and, if so, would the results be more or less precise than FADA?

4 Analysis of Complex Statements

4.1 What is a Complex Statement

All the preceding analyses are predicated on the hypothesis that each operation modifies at most one memory cell. It is not difficult to see that it can be easily extended to cases where an operation modifies a statically bounded set of memory cells.

The situation is more complicated when the language allows the modification of an unbounded set of memory cells by one statement. A case in point is the `read` statement in Fortran:

```
program R
do i = 1,n
  read (*,*) (a(i,j), j = 1,n)
end do
```

Another example is parallel array assignments in languages like Fortran 90, the Perfect Club Fortran (PCF) or HPF. The simplest case is that of the independent `do` loop:

```
program Z                        program ZV
doall (i = 1:n)
  a(i) = 0.0                       a(1:n) = 0.0
end doall
```

Program `Z` is in PCF notation, while `ZV` is in Fortan 90 vector notation.

How are we to handle such idioms in Array Dataflow Analysis? Let us recall the definition (7) of the set of candidate sources:

$$\mathtt{Q}_i(\vec{b}) = \{\vec{a} \mid E_{\mathtt{S}_i}\vec{a} \geq \vec{n}_{\mathtt{S}_i}, \vec{a} \ll \vec{b}, \vec{f}_i(\vec{a}) = \vec{g}(\vec{b})\}.$$

The first problem for a complex statement is that $\vec{a}$ does no longer characterize the values which are created when executing operation $\vec{a}$. We have to introduce auxiliary or *inner* variables to identify each value. In the case of program `R`, for instance, this new variable is in evidence: it is simply the "implicit `do` loop counter", j. The same is true for program `Z`. In the case of program `ZV`, a new counter has to be introduced, let us call it ϕ.

We next have to decide what constraints must be satisfied by these *inner variables*. For the three examples above, these are in evidence from the program text:

$$1 \leq j \leq n$$

for `R`, and:

$$1 \leq \phi \leq n$$

for `ZV`. Objects like:

$$\langle \mathtt{M}[\phi], 1 \leq \phi \leq n \rangle,$$

composed of an array and a subset of the index space of the array, are the *regions* of [CI96]. We will use here generalized regions, of the form:

$$\langle \mathtt{M}[\vec{f}(\vec{a}, \vec{\phi})], A\vec{a} + B\vec{\phi} + \vec{m} \geq 0 \rangle,$$

where $\vec{f}$ is affine, A and B are constant matrices, $\vec{m}$ is a constant vector, and $\vec{a}$ is the vector of the outer variables.

As to the sequencing predicate in (7), it stays the same whatever the type of the candidate statement, since we are supposing here that the corresponding operation is executed in one step. There is, however, a problem with the computation of the latest source, i.e. with the maximum of the candidate set, whose new form is:

$$\mathsf{Q}_i(\vec{b}) = \{\vec{a}, \vec{\phi} \mid E_{\mathsf{S}_i}\vec{a} \geq \vec{n}_{\mathsf{S}_i}, A_i\vec{a} + B_i\vec{\phi} + \vec{m} \geq 0, \vec{a} \ll \vec{b}, \vec{f}_i(\vec{a}, \vec{\phi}) = \vec{g}(\vec{b})\}.$$

We know that sources belonging to different iterations are executed according to lexicographic order, but what of sources belonging to the same iteration? There are several possible situations here.

In the simplest case, that of examples `Z` and `ZT`, the rules of the language insure that there cannot be an output dependence in the `doall` loop or in the vector assignment. This means that $\vec{\phi}$ is uniquely determined by the subscript equations whenever $\vec{a}$ and $\vec{b}$ are known. Hence, there will never be a comparison between sources at the same iteration; we can use any convenient order on the components of $\vec{\phi}$, lexicographic order for instance.

In the case of example `R` there is no such condition on the implicit `do` loops. But, fortunately, the language definition stipulates that these loops are executed in the ordinary way, i.e. in order of lexicographically increasing $\vec{\phi}$, as above.

4.2 ADA in the Presence of Complex Statements

To summarize the preceding discussion, in the presence of complex statements, the first step is the determination of read and modified regions. The usefulness of modified regions is obvious. Read regions delimit the set of memory cells for which sources are to be calculated; their inner variables are simply added as new parameters to the coordinates of the observation statement. In the simple cases we have already examined, the regions can be extracted from a syntactical analysis of the program text. See the next section for a more complicated case.

The analysis then proceeds as in the case of simple statements, the inner variables $\vec{\phi}$ being considered as "virtual" loop counters (which they are in examples `R` and `Z`). The corresponding components are then eliminated or kept, depending on the application, and the direct dependences are combined as above.

4.3 Procedure Calls as Complex Statements

A procedure or function call is a complex statement, as soon as one of its arguments is an array, provided the procedure or function can modify it. This is always possible in Fortran or C. In Pascal, the array argument has to be called by reference. In contrast with the previous examples, one does not know beforehand which parts of which arguments are going to be modified. This information can only be obtained by an analysis of the procedure body itself.

4.3.1 Computing the Input and Output Regions of a Procedure

The case of the output region is the simplest. A cell is modified as soon as there is a an assignment to it in the code. Consider the following assignment statement:

```
for  a⃗ := ···
   M[ f⃗(a⃗)] := ···
```

The associated region is simply:

$$\langle \mathtt{M}[\vec{f}(\vec{\phi})], E\vec{\phi} \geq \vec{n} \rangle.$$

The constraints of the region are given by the bounds of the surrounding loops.

We have to collect all such subregions for a given argument. The result may have redundancies whenever a memory cell is written into several times. This redundancy is harmless, since the write order is not significant outside the procedure. It may however be a source of inefficiency. It can be removed either by polyhedra handling methods or by the following systematic procedure. Suppose we add at the end of the procedure a fictitious observation operation for each cell of each argument[4], and that we compute the corresponding source. The result is a quast which depends on the subscripts of the array cell, $\vec{\phi}$. For each leaf whose value is not $\perp$, we may construct a subregion:

$$\langle \mathtt{M}[\vec{\phi}], C(\vec{\phi}) \rangle,$$

where C is the context of the distinguished leaf. The result will have no redundancy.

The computation of the input region is more difficult. Notice first that it is not the same thing as the read region, as shown by the elementary example:

```
1 : x := ...;
2 : ... := x;
```

`x` is read but is not in the input region, since its entry value is killed by $\mathtt{S}_1$. Computing the output region as accurately as possible is important, since a source is to be computed for each of its cells in the calling routine. Redundancies will induce useless computation; inaccuracies generate spurious dependences and lessen parallelism. The solution is to compute the earliest access to each cell of each argument of the procedure. One collect all accesses to a cell in the body of the procedure, whether reads or writes. This gives a set of candidates, of which one computes the lexicographic minimum using the same technology as in the source computation[5]. The resulting quast gives the earliest access to each argument cell as a function of its subscripts. If the access is a read, the cell is in the input region. If it is a write, it is not. Lastly, if the leaf is $\perp$, then the cell is not used in the procedure. Subregions of the input are associated to read leaves in the quast, and are constructed in the same way as in the case of the output region.

If the procedure is not a static control program, we have to use techniques from FADA when computing the input and output regions. Fuzziness in the input region is not important. It simply means the loss of some parallelism. Fuzziness in the output region is more critical, and may preclude Dataflow

[4] e.g., a `print` statement.

[5] Note that there is a subtle point in the use of rule 3 for this problem. We may have to compare an operation to itself, if it includes both a read and a write to the same cell. Obviously, the read always occurs *before* the write. In the line:

```
s := s + 1;
```

the read of `s` occurs before the write, hence `s` *is* in the input region.

Analysis of the calling routine, for reasons which have been explained above (see Sect. 3.1).

This analysis gives the input and output regions of a procedure, but they are expressed in term of the procedure formal arguments. To proceed to the dataflow analysis of the calling routine, we have to translate these regions in term of the caller variables, i.e. in term of the actual arguments of the procedure. This translation is easy in Pascal, since actual and formal parameters must have exactly the same type: one has simply to change the name of formal arrays to actual arrays in each subregion. In the case of Fortran or C, where the typing is less strict, one has to exhibit the addressing function (or linearization function) of the formal and actual arrays. The relation between actual and formal subscripts is obtained by writing that the two array accesses are to the same memory cell, and that the subscripts are within the array bounds. In simple cases, one may find closed formulas expressing one of the subscript set in term of the other. If the bounds are explicitly given numbers, the problem can be solved by ILP. There remains the case of symbolic array bounds, in which one has to resort to *ad hoc* methods which are not guaranteed to succeed [CI96].

4.3.2 Organizing the Analysis

In Fortran, procedures cannot be recursive. Hence, one may draw a call tree. The interprocedural dataflow analysis can be done bottom up. Leaves call no procedure, hence their regions can be calculated without difficulty. If the input and output regions of all called procedures are known, then the input and input regions of the caller can be computed. When all input and output regions are known, then array dataflow analysis can be executed independently for all procedures.

Input and output regions can be stored in a library, along with other information about the procedure, such as its type and the type of its arguments. Care should be taken, however, that the region information is not intrinsic to the procedure and has to be computed again whenever the procedure or one of the procedures it calls (directly or indirectly) is modified.

Input and Output regions calculation for recursive procedures is an open problem. It is probably possible to set it up as a fixpoint calculation, but all technical details (monotony, convergence, complexity, ...) are yet to be designed.

[CI96] gives another method for computing input and output regions. Regions are approximated by convex polyhedra, and dataflow equations are used to propagate regions through the program structure. The overall organization of the computations is the same as the one given here.

5 Applications of ADA and FADA

All applications of ADA and FADA derives from two facts:

- The method is static: it can be used at compile time, without any knowledge besides the program text.
- The result is a closed representation of a dynamic phenomenon: the creation and use of values as the execution of the program proceeds.

One may in fact consider that the dataflow of a program is one possible representation of its semantics. If this is stipulated, then ADA is a way of extracting a semantics from a program text. FADA gives the same information, but with lesser precision. Hence, ADA and FADA are useful as soon as one needs to go beyond the "word for word" or "sentence for sentence" translation that is done by most ordinary compilers. Case in points are program understanding and debugging, all kinds of optimization including parallelization, and specially array expansion and privatization.

5.1 Program Comprehension and Debugging

A very simple application of ADA and FADA is the detection of uninitialized variables. Each occurrence of a $\perp$ in a source indicates that a memory cell is read but that there is no write to this cell *before* the read. If we are given a complete program, this clearly suggests a programming error. The program has to be complete: it should include all statements which can set the value of a variable, including `read` statements, initializations, and even hidden initialization by, e.g., the underlying operating system. Note that the presence of a $\perp$ in a source is not absolute proof of an error. For instance, in:

```
x := y * z;
```

`y` may be uninitialized if one is sure that `z` is zero. In the case of ADA, the access to an uninitialized variable may be conditional on the values of structure parameters. An example is:

```
    do i = 1,n
1      s = ...
    end do
2   x = s
```

The source of `s` in S_2 is $\mathbf{if}\ n \geq 1\ \mathbf{then}\ \langle 1, n, 1\rangle\ \mathbf{else}\ \perp$. There is an error if $n < 1$. This situation may be explicitly forbidden by the program definition, or, better, by a test on n just after its defining statement. One may use any number of techniques to propagate the test information through the program (see e.g. [JF90]) and use it to improve the analysis.

The situation is more complicated for FADA. The presence of a $\perp$ in a source indicates that, for some choice of the intractable predicates, an access to an uninitialized variable may occur. But this situation may be forbidden by facts about the intractable predicates that we know nothing about, or that we are not clever enough to deduce from the program text. In this situation, one should either shift to more precise analyses (for instance use semantical knowledge), or just check the program by hand to show that the error never occurs.

The same technology which is used for ADA can be reused for checking the correctness of array accesses. The results take the form of conditions on the structure parameters for the subscripts to be within bounds. These conditions can be tested once and for all as soon as the values of structure parameters are known, giving a far more efficient procedure than the run-time tests which are generated by some Pascal compilers.

The knowledge of exact sources allows the translation of a program into a system of recurrence equations (SRE):

$$\vec{a} \in \mathcal{D}_i : v_i[\vec{a}] = \mathcal{E}(\ldots, v_k[f_{ik}(\vec{a})], \ldots), i = 1, n, \tag{38}$$

where $\mathcal{D}_i$ is the domain of the equation (a set of integer vectors), v_i and v_k are "variables" (functions from integer vectors to an unspecified set of values), and the f_{ik} are dependence functions. $\mathcal{E}$ is an arbitrary expression, most of the time a conditional. Systems of recurrence equations where introduced in [KMW67]. Concrete representations of such systems are used as the starting point of systolic array synthesis (see for instance [LMQ91]).

To transform a static control program into an SRE, first assign a distinct variable to each assignment statement. The domain of v_i associated to Statement $\mathtt{S}_i$ is the iteration domain of $\mathtt{S}_i$, and the left hand side of the corresponding equation is simply $v_i[\vec{a}]$ where $\vec{a}$ scans $\mathcal{D}_i$. The expression $\mathcal{E}$ is the right hand side of the assignment, where each memory cell is replaced bu its source. If some of the sources include $\perp$'s, the original arrays of the source program are to be kept as non mutable variables and each $\perp$ is to be converted back to the original reference (see [Fea91] for details).

As an example of semantics extraction, consider the program piece:

```
for i := 1 to n do m[i] := m[i+1]
```

The source of `m[i+1]` is $\perp$. The equivalent SER is:

$$v[i] = m[i+1], i = 1, n.$$

In the case of:

```
fo i := 1 to n do m[i] := m[i-1]
```

the source of `m[i-1]` is $\textbf{if } i > 1 \textbf{ then } \langle 1, i-1, i\rangle \textbf{ else } \perp$. The equivalent SER is:

$$v[i] = \textbf{if } i > 1 \textbf{ then } v[i-1] \textbf{ else } m[i-1].$$

The first recurrence clearly represents a "left shift" while the second one is a rightward propagation of $v[0]$.

An SER is is a mathematical object which can be submitted to ordinary reasoning and transformations. One can say that an SER conveys the semantics of the source program, and ADA is this sense is a semantics extractor. The process can be pursued one step further by recognizing scans and reductions [RF93].

One can also think of an SER as a (Dynamic) Single Assignment program. Most of the time, the memory needs of a DSA program are prohibitive. It is best to think of a DSA program (or of the associated SER) as an intermediate step in the compilation process.

The results of FADA are to be thought of as an approximate semantics. It is much more difficult to convert them into something approaching a well defined mathematical object. One has to resort to dynamically gathered information to select the real source among the elements of a source set. The reader is referred to [GC95] for details.

5.2 Parallelization

The main use of source information is in the construction of parallel programs. Two operation in a program are in (data) dependence if they share a memory cell and one of them at least modifies it. Dependent operations must be executed sequentially. Other operations can be executed concurrently. Dependences are classified as flow dependences, in which a value is stored for later use, and anti- and output dependences, which are related to the sharing of a memory cell by two unrelated values. The later type of dependence can be removed by data expansion, while flow dependences are inherent to the underlying algorithm. It follows that maximum parallelism is obtained by taking into account the source relation only: an operation must always be executed after all its sources.

These indications can be formalized by computing a schedule, i.e. a function which gives the execution date of each operation in the program. All operations which are scheduled at the same time can be executed in parallel. For reasons which are too complicated to explain here (see [Fea89]), one does not have to use the exact execution time of each operation when computing a schedule, provided the amount of parallelism is large. One may suppose that all operations take unit time. The schedule θ must then satisfy:

$$\theta(u) \geq \theta(\varsigma(u)) + 1, \tag{39}$$

for all operations u in the case of ADA, and

$$\forall v \in \Sigma(u) : \theta(u) \geq \theta(v) + 1, \tag{40}$$

in the case of FADA. These systems of functional inequalities have in general many solutions. For reasons of expediency, one usually selects a particular type of solution (in fact, the solutions which are affine functions of the loop counters) and solve either (39) or (40) by linear programming. The reader is referred to [Fea92a, Fea92b] for details of the solution method.

Some programs do not have affine schedules – i.e. the associated linear programming problem proves unfeasible. In that case, one must resort to *multidimensional schedules*, in which the value of θ is a d dimensional vector. The execution order is taken to be lexicographic order. Suppose we are dealing with a N-deep loop nest. Having a d dimensional schedule means that the parallel program will have d sequential loops enclosing $N - d$ parallel loops. Using such schedules may be necessary because the source program has a limited amount of parallelism, or because we are using overestimates of the dependences from FADA, or simply because we want to adapt a schedule to a parallel processor by artificially reducing the amount of parallelism.

5.3 Array Expansion and Array Privatization

It is easy to see that the degree of parallelism of a program is closely related to the size of its working memory (the part of its memory space the program can write into), since independent operations must write into distinct memory cells. Consider a loop nest that we hope to execute on P processors. This is only possible if the nest uses at least P cells of working memory. Parallelization may thus be prevented by too small a memory space, since programmers have a natural tendency to optimize memory. *A contrario*, a parallelizer may have to enlarge the working memory to obtain an efficient parallel program.

This can be done in two ways. Consider for instance the kernel of a matrix-vector code:

```
    for i := 1 to n do
    begin
1 :   s := 0;
      for j := 1 to n do
2 :     s := s + a[i,j]*x[j]
    end
```

The working memory is `s` of size one, hence the program is sequential. The first possibility is to *privatize* `s`, i.e. to provide one copy of `s` per processor. How do we know that this transformation is allowed? Observe that our objective here is to find parallel loops. If the i loop, for instance, is parallelized, distinct iterations may be executed by distinct processors. Since each processor has its copy of `s`, this means there must not be any exchange of information through `s` between iterations of the i loop. The same is true for the j loop if we decide to parallelize it. Now consider the source of `s` in statement 2. It is easily computed to give:

$$\varsigma(1, i, 2, j, 1) = \textbf{if } j > 1 \textbf{ then } \langle 1, i, 2, j-1, 1\rangle \textbf{ else } \langle 1, i, 1\rangle.$$

It is clear that there is no information flow from iteration i to $i', i \neq i'$. There is, on the contrary, a data flow from iteration $j-1$ to iteration j. This show both that the i loop is parallel and that `s` must be privatized, giving:

```
    forall i := 1 to n do
    begin
      s : private real;
1 :   s := 0;
      for j := 1 to n do
2 :     s := s + a[i,j]*x[j]
    end
```

This method generalizes to array privatization. For another approach, see [TP94].

There is however another method, which is to resort to array expansion instead of array privatization. The first idea that comes to mind is to use the Dynamic Single Assignment version of the program, thus insuring that all output dependences are satisfied. The result in the above case is:

```
    forall i := 1 to n do
    begin
1 :   s1[i] := 0;
      for j := 1 to n do
2 :     s2[i,j] := (if j > 1
                    then s2[i,j-1]
                    else s1[i]) + a[i,j]*x[j]
    end
```

Notice however that while the original memory size was $O(1)$, it is now $O(n^2)$, the amount of parallelism being only $O(n)$. The degree of expansion is clearly too large. It is possible, by analyzing the life span of each value in the program, to find the minimum expansion for a given schedule [LF97]. In the present case, one finds:

```
    forall i := 1 to n do
    begin
1 :   s[i] := 0;
      for j := 1 to n do
2 :     s[i] := s[i] + a[i,j]*x[j]
    end
```

Suppose we are using P processors. This may still be too much if n is much larger than P, as it should for efficiency sake. The solution is to adjust the schedule for the right amount of parallelism. The optimal schedule is $\theta(1, i, 2, j, 1) = j$ which should be replaced by the two dimensional version:

$$\theta(1, i, 2, j, 1) = \left(\begin{array}{c} j \\ i \bmod P \end{array} \right).$$

The resulting program is[6]:

```
    forall ii := 1 to P do
      for k := ii to n by P do
      begin
1 :     s[ii] := 0;
        for j := 1 to n do
2 :       s[ii] := s[ii] + a[k,j]*x[j]
    end
```

The amount of expansion is now exactly equal to the amount of parallelism.

6 Conclusions

Let us take a look at was has been achieved so far. We have presented a technique for extracting semantics information from sequential imperative programs at compile time. The information we get is exact and, in fact, exhaustive in the case of static control programs. In the case of less regular programs, we get approximate results, the degree of approximation being in exact proportion of the irregularity of the source code.

Array Dataflow information has many uses, some of which have been presented here, in program analysis and checking, program optimization and program parallelization. There are other applications, some of which have not been reported here due to lack of space [RF93], while others are still awaiting further developments: consider for instance the problem of improving the locality of sequential and parallel codes.

There are still many problems in the design of Array Dataflow Analyses. For instance, what is the relation between FADA and methods based on Abstract Interpretation? What is the best way of taking advantage of semantical information about the source program? Can we extend Dataflow Analysis to other data structures, e.g. trees? All these questions will be the subject of future research.

[6] For simplicity, we have supposed that P divide n.

A Appendix : Mathematical Tools

The basic reference on linear inequalities in rationals or integers is the treatise [Sch86].

A.1 Polyhedra and Polytopes

There are two ways of defining a polyhedron. The simplest one is to give a set of linear inequalities:

$$A\vec{x} + \vec{a} \geq 0.$$

The polyhedron is the set of all $\vec{x}$ which satisfies these inequalities. A polyhedron can be empty – the set of defining inequalities is said to be *unfeasible* – or unbounded. A bounded polyhedron is called a polytope.

The basic property of a polyhedron is *convexity*: if two points $\vec{a}$ and $\vec{b}$ belong to a polyhedron, then so do all convex combinations $\lambda\vec{a} + (1 - \lambda)\vec{b}, 0 \leq \lambda \leq 1$. Conversely, it can be shown that any polyhedron can be generated by convex combinations of a finite set of points, some of which – rays – may be at infinity. Any polyhedron is generated by a minimal set of vertices and rays.

There exist non-polynomial algorithms for going from a representation by inequalities to a representation by vertices and rays and vice-versa. Each representation has its merits: for instance, inequalities are better for constructing intersections, while vertices are better for convex unions[7].

The basic algorithms for handling polyhedra are feasibility tests: the Fourier-Motzkin cross-elimination method [Fou90] and the Simplex [Dan63]. The interested reader is referred to the above quoted treatise of Schrijver for details. Both algorithms prove that the object polyhedron is empty, or exhibit a point which belongs to it. For definiteness, this point is generally the lexicographic minimum of the polyhedron. In the case of the Fourier-Motzkin algorithm, the construction of the exhibit point is a well separated phase which is omitted in most cases.

Both the Fourier-Motzkin and the Simplex are variants of the Gaussian elimination scheme, with different rules for selecting the pivot row and column. Theoretical results and experience have shown that the Fourier-Motzkin algorithm is faster for small problems (less than about 10 inequalities), while the Simplex is better for larger problems.

A.2 Z-modules

Let $v_1, \ldots, v_n$ be a set of linearly independent vectors of $\mathbb{Z}^n$ with integral components. The set:

$$\mathcal{L}(v_1, \ldots, v_n) = \{\mu_1 v_1 + \ldots + \mu_n v_n \mid \mu_i \in \mathbb{Z}\}$$

is the Z-module generated by $v_1, \ldots, v_n$. The set of all integral points in $\mathbb{Z}^n$ is the Z-module generated by the canonical basis vectors (the canonical Z-module).

Any Z-module can be characterized by the square matrix V of which $v_1, \ldots, v_n$ are the column vectors. We will use the notation $\mathcal{L}(V)$ for $\mathcal{L}(v_1, \ldots, v_n)$. However, many different matrices may represent the same Z-module. A square

[7] Notice that while the intersection of two polyhedra is a polyhedron, their union is not.

matrix is said to be unimodular if it has integral coefficients and if its determinant is ± 1. Let U be a unimodular matrix. It is easy to prove that V and VU generate the same lattice.

Conversely, it can be shown that any non-singular matrix V can be written in the form $V = HU$ where U is unimodular and H has the following properties:

- H is lower triangular,
- All coefficients of H are positive,
- The coefficients in the diagonal of H dominate coefficients in the same row.

H is the Hermite normal form of V. Two matrices generate the same Z-module if they have the same Hermite normal form. The Hermite normal form of a unimodular matrix is the identity matrix, which generates the canonical Z-module.

Computing the Hermite normal form of an $n \times n$ matrix is of complexity $O(n^3)$, provided that the integers generated in the process are of such size that arithmetic operations can still be done in time $O(1)$.

A.3 Z-polyhedra

A Z-polyhedron is the intersection of a Z-module and a polyhedron:

$$F = \{\vec{z} \mid \vec{z} \in \mathcal{L}(V), A\vec{z} + \vec{a} \geq 0\}.$$

If the context is clear, and if $\mathcal{L}(V)$ is the canonical Z-module ($V = I$), it may be omitted in the definition.

The basic problem about Z-polyhedra is the question of their emptiness or not. For canonical Z-polyhedra, this is the linear integer programming question [Sch86, Min83]. Studies in static program analysis use either the Omega test [Pug91] which is an extension of Fourier-Motzkin, or the Gomory cut method, which is an extension of the Simplex [Gom63].

Both the Omega test and the Gomory cut method are inherently non polynomial algorithms, since the integer programming problem is known to be NP-complete.

A.4 Parametric Problems

A linear programming problem is parametric if some of its elements – e.g. the coefficients of the constraint matrix or those of the economic function – depend on parameters. In problems associated to parallelization, it so happens that constraints are often linear with respect to parameters. In fact, most of the time we are given a polyhedron $\mathcal{P}$:

$$A \begin{pmatrix} \vec{x} \\ \vec{y} \end{pmatrix} + \vec{a} \geq 0$$

in which the variables have been partitioned in two sets, the unknowns: $\vec{x}$, and the parameters: $\vec{y}$. Setting the values of the parameters to $\vec{p}$ is equivalent to considering the intersection of $\mathcal{P}$ with the hyperplane $\vec{y} = \vec{p}$, which is also a polyhedron. In a parametric problem, we have to find the lexicographic minimum of this intersection as a function of $\vec{p}$.

The Fourier-Motzkin method is "naturally" parametric in this sense. One only has to eliminate the unknowns from the last component of $\vec{x}$ to the first. When this is done, the remaining inequalities give the conditions that the parameters must satisfy for the intersection to be non empty. If this condition is verified, each unknown is set to its minimum possible value, i.e. to the maximum of all its lower bounds. Let $C\vec{y} + \vec{c} \geq 0$ be the resulting inequalities after elimination of all unknowns. The parametric solution may be written:

$$\min_{\ll}(\mathcal{P} \cap \{\vec{y} = \vec{p}\}) = \textbf{if } C\vec{p} + \vec{c} \geq 0 \textbf{ then } \begin{pmatrix} \max(f(\vec{p}), \ldots, g(\vec{p})) \\ \cdots \\ \max(h(\vec{p}), \ldots, k(\vec{p})) \end{pmatrix} \textbf{ else } \perp$$

where $\perp$ is the undefined value and the functions $f, \ldots, k$ are affine.

The simplex also relies on linear combinations of the constraint matrix rows, which can be applied without difficulty in the parametric case. The only difficulty lies in the choice of the pivot row, which is such that its constant coefficient must be negative. Since this coefficient depends in general on the parameters, its sign cannot be ascertained; the problem must be split in two, with opposite hypotheses on this sign. These hypotheses are not independent; each one restricts the possible values of the parameters, until inconsistent hypotheses are encountered. At this point, the splitting process stops. By climbing back the problem tree, one may reconstruct the solution in the form of a multistage conditional. The advantage of the parametric Simplex over the Fourier-Motzkin algorithm is that it can be extended to the all-integer case. Parametric Gomory cuts can be constructed by introducing new parameters which represent integer quotients. The reader is referred to [Fea88b] for an implementation of these ideas in the Parametric Integer Programming (PIP) algorithm.

References

[AK87] J. R. Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM TOPLAS*, 9(4):491–542, October 1987.

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

[BCF97] Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *ACM Int. Conf. on Supercomputing*, St Malo, France, July 1988.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symp. on Principle and Practice of Programming Languages*, pages 238–252. ACM, 1977.

[CI96] Béatrice Creusillet and François Irigoin. Interprocedural array regions analyses. *Int. J. of Parallel Programming*, 24(6):513–546, 1996.

[Dan63] G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.

[Fea88a] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

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

[Fea89] Paul Feautrier. Asymptotically efficent algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized System*, pages 273–284. IFIP WG 10.3, North-Holland, December 1989.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Flo67] Robert J. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science.* AMS, 1967.

[Fou90] J. B. J. Fourier. *Oeuvres de Fourier, Tome II.* Gauthier-Villard, Paris, 1890.

[GC95] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In *Euro-Par95*, Stockholm, Sweden, Aug 1995.

[GLL95] Jungie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, December 1995.

[Gom63] R. E. Gomory. An algorithm for integer solutions to linear programs. In R. L. Graves and P. Wolfe, editors, *Recent Advances in Math. Programming*, chapter 34, pages 269–302. Mac-Graw Hill, New York, 1963.

[HT94] C. Heckler and L. Thiele. Computing linear data dependencies in nested loop programs. *Parallel Processing Letters*, 4(3):193–204, 1994.

[JF90] Pierre Jouvelot and Paul Feautrier. Parallélisation Sémantique. *Informatique théorique et Applications*, 24:131–159, 1990.

[KMW67] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[KP96] Induprakas Kodokula and Keshav Pingali. Transformations for imperfect nested loops. In *Supercomputing*, 1996.

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

[Les96] Arnauld Leservot. *Analyse Interprocédurale du flot des données.* PhD thesis, Université Paris VI, March 1996.

[LF97] Vincent Lefebvre and Paul Feautrier. Storage management in parallel programs. In IEEE Computer Society, editor, *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 181–188, Londres (England), January 1997.

[LMQ91] Hervé Leverge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993.

[Min83] Michel Minoux. *Programmation Mathématique, théorie et algorithmes.* Dunod, Paris, 1983.

[Pug91] William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, 1991.

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

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag LNCS 768, August 1993.

[RF93] Xavier Redon and Paul Feautrier. Detection of reductions in sequential programs with loops. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Procs. of the 5th Int. Parallel Architectures and Languages Europe*, pages 132–145. LNCS 694, June 1993.

[Sch86] A. Schrijver. *Theory of linear and integer programming.* Wiley, NewYork, 1986.

[SJ77] N. Suzuki and D. Jefferson. Verification decidability of Pressburger array programs. In *Procs. of a conf. on TCS*, Waterloo, 1977.

[TIF86] Rémi Triolet, François Irigoin, and Paul Feautrier. Automatic parallelization of FORTRAN programs in the presence of procedure calls. In Bernard Robinet and R. Wilhelm, editors, *ESOP 1986, LNCS 213*. Springer-Verlag, 1986.

[TP94] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Proc. of the 7th Workshop on Languages and Compilers for Parallel Computers, LNCS 892*, 1994.

[Won95] David G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, U. of Maryland, 1995.