Interprocedural Array Region Analyses

Béatrice CREUSILLET, François IRIGOIN*

Centre de Recherche en Informatique, École des mines de Paris
35, rue Saint-Honoré, F-77305 FONTAINEBLEAU Cedex FRANCE

Abstract. Many program optimizations require exact knowledge of the sets of array elements that are referenced in or that flow between statements or procedures. Some examples are array privatization, generation of communications in distributed memory machines, or compile-time optimization of cache behavior in hierarchical memory machines.
Exact array region analysis is introduced in this article. These regions exactly represent the effects of statements and procedures upon array variables. To represent the flow of these data, we also introduce two new types of array region analyses: IN and OUT regions.
The intraprocedural propagation is presented, as well as a general linear framework for interprocedural analyses, which handles array reshapes.
The intra- and inter-procedural propagation of array regions is implemented in PIPS, the interprocedural parallelizer of FORTRAN programs developed at École des mines de Paris.
Keywords: interprocedural analysis, array data flow analysis, array regions, array reshaping.

1 Introduction

The efficient compilation of scientific programs for massively parallel machines or hierarchical memory machines requires advanced program optimizations to deal with memory management issues. For instance, Blume and Eigenmann[6] have shown that array privatization could greatly enhance the amount of potential parallelism in sequential programs. This technique basically aims at discovering array sections that are used as temporaries in loops, and can thus be replaced by local copies on each processor. An array section is said to be privatizable in a loop if each read of an array element is preceded by a write in the same iteration, and several different iterations may access each privatized array element[24, 34]. Solving such problems requires a precise intra- and inter-procedural analysis of array data flow, that is to say how individual array element values are defined and used (or *flow*) during program execution.

A recent type of analysis[7, 16] has opened up wide perspectives in this area: It provides an *exact* analysis of array data flow, originally in monoprocedural programs with static control. This last constraint has since been partially removed[25, 12], at the expense of accuracy. A partial interprocedural

* E-mail: {creusillet,irigoin}@cri.ensmp.fr

extension[23] has also been defined, but only in a static control framework. Furthermore the complexity of the method makes it useless on large programs.

Another approach is to compute conservative summaries of the effects of statements and procedure calls on sets of array elements[33, 9]. Their relatively weak complexity (in practice) allows the analysis of large programs. But since these analyses are flow insensitive, and since they do not precisely take into account the modifications of the values of integer scalar variables, they are not accurate enough to support powerful optimizations.

In PIPS[21], the interprocedural parallelizer of FORTRAN programs developed at École des mines de Paris, we have extended Triolet's array regions[33] (which are array element sets described by convex polyhedra) to compute summaries that *exactly* represent the effects of statements and procedures on sets of array elements[3], whenever possible; whereas the regions originally defined by Triolet were *over-approximations* of these effects.

The resulting *exact* READ and WRITE regions were found necessary by Coelho[10, 11] to efficiently compile HPF. However, they cannot be used to compute array data flow, and are thus insufficient for optimizations such as array privatization.

We therefore introduce two new types of exact regions: for any statement or procedure, IN regions contain its imported array elements, and OUT regions represent its set of live array elements.

The possible applications are numerous. IN and OUT regions are already used in PIPS to privatize array sections[3], and we intend to use them for memory allocation when compiling signal processing specifications based on dynamic single assignment. In massively parallel or heterogeneous systems, they can also be used to compute the communications before and after the execution of a piece of code. For a hierarchical memory machine, they provide the sets of array elements that are used or reused, and hence could be prefetched (IN regions) or kept (OUT regions) in caches; the array elements that do not appear in these sets are only temporaries, and should be handled as such. In fault-tolerant systems where the current state is regularly saved by a software component (*checkpointing*[22]) IN or OUT regions could provide the set of elements that will be used in further computations, and thus could be used to reduce the amount of data to be saved. Examples of other applications are software specification verification or compilation of out-of-core computations[28].

To support the exactness of the analysis, an accurate interprocedural translation is needed. However, by examining the Perfect Club Benchmarks[5], we found out that the existing methods for handling array reshapes were insufficient. We propose in this paper a general linear framework that handles array reshaping in most cases, including when the arrays are not of the same type, or belong to a `COMMON` which does not have the same data layout in the caller and the callee.

This paper is organized as follows. Section 2 presents a motivating example that highlights the mains difficulties of region computation. Some necessary background is shortly reviewed in Section 3. Section 4 presents array regions and their operators. The intraprocedural propagation of READ, WRITE, IN and

OUT regions is detailed in Section 5. The interprocedural propagation is then separately described in Section 6. And Section 7 reviews the related work.

2 Motivating Example

To illustrate the main features of the intraprocedural computation of READ-WRITE, IN and OUT regions along this article, we consider the contrived program of Figure 1. The goal is to privatize array WORK.

```
K = FOO()                                          SUBROUTINE INC1(I)
DO I = 1,N                                         I = I + 1
   DO J = 1,N                                      END
      WORK(J,K) = J + K
   ENDDO
   CALL INC1(K)
   DO J = 1,N
      WORK(J,K) = J*J - K*K
      A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
   ENDDO
ENDDO
```

Fig. 1. Sample program.

The condition is that any iteration of the I loop neither imports nor exports any element of the array WORK. In other words, if there is a read reference to an element of WORK, it has been previously initialized in the same iteration, and it is not reused in the subsequent iterations (we assume that the array WORK is not used anymore after the I loop).

There are two main difficulties in our example. First, different elements of WORK are referenced in several instructions. We shall need several operators to manipulate the regions representing these references, and compute the solutions to data-flow problems, e.g. union, intersection or difference. Second, these references, and thus their representations, depend on the value of the variable K, which is unknown at the entry of the I loop, and is modified by the call. We need an operator to obtain representations that depend on the same value of K, and hence can be combined.

The next two sections present the techniques used to perform the analysis of our example.

3 Language, Transformers and Preconditions

In PIPS[21] the parallelization process is divided into several phases, either analyses (e.g. transformers, preconditions, array regions) or program transformations (e.g. dead code elimination, loop transformations). Most analyses also consist of

two types of propagation: *intra-* and *inter*-procedural propagations. This section describes the general mechanisms involved in both types of propagation, as well as two analyses performed in PIPS and whose results are used to compute array regions.

3.1 Language, HCFG and call graph

Intraprocedural propagations are performed on the *hierarchical control flow graph*[21] (HCFG) of the routines. This graph bears some resemblance to the abstract syntax tree of the program: Most nodes of the HCFG correspond to the FORTRAN language control structures (`DO` loop, `IF`, sequence of instructions, assignment, call, ...), except for the unstructured parts of the program (when `GOTO`s or `STOP`s are used) which are modeled by standard control flow graphs.

An example of such a graph is provided in Figure 2. The nodes are represented by rectangles. The biggest node on the left is a sequence of several instructions, represented by sub-nodes. One of these sub-nodes is itself a `DO` loop node. Its inner node is a sequence of two instructions.

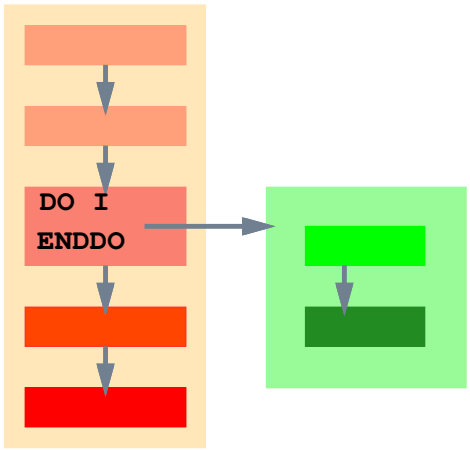


Fig. 2. Example of HCFG.

In this article, we only consider assignments, `DO` loops with unit increments, sequences of instructions, and procedure calls. The other constructs, in particular `IF` constructs, are not considered here, because it would not provide useful insights to the reader. However, the implementation of array region computation

in PIPS covers the whole FORTRAN standard[1], with a few minor exceptions[2] which can easily be avoided.

Bottom-up analyses propagate their results towards the root of the HCFG (entry node of the procedure): the deepest nodes are first analyzed, and the results are used at the upper level to form another solution which is similarly propagated. On the contrary, top-down analyses propagate the solutions toward the leaves of the tree: the solution for the inner nodes are computed from the solutions at the upper level.

Interprocedural propagations are performed on the program *call graph*. This graph is assumed acyclic, according to the FORTRAN standard[1] which prohibits recursive function calls. Analyses can be performed bottom-up or top-down. In the first case, the intraprocedural analysis of the deepest procedures is performed first; the information at the root node of their HCFG is then propagated to the various call sites by translating formal parameters into actual ones; the callers are then intraprocedurally analyzed using the preceding interprocedural solutions, and so on. On the contrary, in a top-down propagation, the main program is first intraprocedurally analyzed starting from its entry point; the solutions at each call site are then propagated to the callees by translating actual parameters into formal ones; when there are several call sites for one procedure, the solutions are gathered into a unique summary, to limit time and space complexity.

Whether the analysis is bottom-up or top-down, each node of the HCFGs or of the call graph is traversed only once. The complexity of an analysis thus mostly depends on the complexity of the operations performed at each node. As will be shown later, many semantical analyses in PIPS (transformers, preconditions and array regions) rely on convex polyhedra. Most operators have a theoretical exponential complexity, but the practical complexity often is polynomial. Furthermore the exponential speed improvement of computers renders these analyses fast enough to perform them on real life programs.

3.2 Transformers and preconditions

Two auxiliary analyses are of interest in the remainder of this paper: *transformers* and *preconditions*[20].

Transformers abstract the effects of instructions upon the values of integer scalar variables by giving an affine approximation of the relations that exist between their values before and after the execution of a statement or procedure call. In equations they are designated by T, whereas in programs they appear under the form `T(args) {pred}`, where `args` is the list of modified variables, and `pred` gives the non trivial relations existing between the initial values (suffixed by `#init`) and the new values of variables. Figure 3 shows the transformers of our working example.

[2] `ENTRY`, `BLOCKDATA`, `ASSIGN` and assigned `GOTO`, computed `GOTO`, multiple `RETURN`, substring operator (`:`), Hollerith character chains, statement functions, and complex constants (which are replaced by a call to `CMPLX`); `COMMON` declarations must also appear after all type declarations.

```
C  P() {}
C  T(K) {}
    K = FOO()
C  P(K) {}
C  T(K) {K==K#init+I-1}
    DO I = 1,N
C  P(I,K) {1<=I<=N}
       DO J = 1,N
C  P(I,J,K) {1<=I<=N, 1<=J<=N}
          WORK(J,K) = J + K
       ENDDO
C  P(I,K) {1<=I<=N}
C  T(K) {K==K#init+1}
       CALL INC1(K)
       DO J = 1,N
C  P(I,J,K) {1<=I<=N, 1<=J<=N}
          WORK(J,K) = J*J - K*K
          A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
       ENDDO
    ENDDO
```

Fig. 3. Transformers and preconditions.

Preconditions are predicates over integer scalar variables. They hold just before the execution of the corresponding instruction. In Figure 3, they appear as `P(vars) {pred}`, where `vars` is the list of modified variables since the beginning of the current routine, because preconditions abstract the effects of the routine from its entry point to the current instruction.

Transformers are propagated upward, while preconditions are propagated downward. And if T_1 and P_1 correspond to the instruction S_1, and P_2 to the instruction S_2 immediately following S_1, then $P_2 = T_1(P_1)$.

4 Regions: Definitions and Operators

An array region is a set of array elements described by a convex polyhedron containing equalities and inequalities[33]: they link the *region parameters* (or ϕ variables) that represent the array dimensions, to the values of the program integer scalar variables. Two other characteristics are also of interest:

- the *type* of the region: READ (`R`) or WRITE (`W`) to represent the effects of statements and procedures; IN and OUT to represent the flow of array elements;
- the *approximation* of the region: `EXACT` when the region exactly represents the requested set of array elements, or `MAY` or `MUST` if it is an over- or under-approximation (`MUST` $\subseteq$ `EXACT` $\subseteq$ `MAY`); in the rest of the paper, we only

consider EXACT and MAY regions; in previous papers[15, 14] MUST was unfortunately used to mean EXACT.

For instance, the region:

$$\texttt{<A(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{}\phi_1\texttt{==I, }\phi_1\texttt{==}\phi_2\texttt{\}>}$$

where the region parameters ϕ_1 and ϕ_2 respectively represent the first and second dimensions of A, corresponds to an assignment of the element A(I,I).

In order to summarize array accesses at each level of the HCFG (to avoid space complexity), and to propagate the summaries along control flow paths, we need several operators such as union, intersection and difference, and more specific unary operators.

Union The union operator is used to merge two elementary regions. Since the union of two convex polyhedra is not necessarily a convex polyhedron, the approximate operator $\bar{\cup}$ we use is the convex hull. The resulting region may thus contain array elements that do not belong to the original regions; in this case[3], it is a MAY region. The third column in Table 1 gives the approximation of the resulting region against the characteristics of the initial regions.

R_1	R_2	$R_1 \bar{\cup} R_2$	$R_1 \cap R_2$	$R_1 \ominus R_2$
EXACT	EXACT	EXACT iff $\equiv R_1 \cup R_2$	EXACT	$\overline{\bigcup}(R_1 \cap \bar{R}_2)$, EXACT iff $\equiv R_1 - R_2$
EXACT	MAY	EXACT iff $R_2 \subseteq R_1$	MAY	R_1, EXACT iff $R_1 \cap R_2 = \varnothing$
MAY	EXACT	EXACT iff $R_1 \subseteq R_2$	MAY	$\overline{\bigcup}(R_1 \cap \bar{R}_2)$, MAY
MAY	MAY	MAY	MAY	R_1, MAY

(all the operators and tests used in this table are implemented in PIPS)

Table 1. Binary operators on regions

Intersection The intersection of two convex polyhedra is a convex polyhedron. It follows that the intersection of two EXACT regions is an EXACT region. A more complete description of this operator is given in Table 1, Column 4.

Difference The difference of two convex polyhedra is not necessarily a convex polyhedron. The chosen operator $\ominus$ may give an over-approximation of the actual difference of the original regions. Its features are described in Table 1, Column 5. For instance, when the original regions are EXACT regions, a first step computes $R_1 \cap \bar{R}_2$; the result is a list of regions[3]; these regions are then merged using $\overline{\bigcup}$, an extension of $\bar{\cup}$ to union of lists.

[3] The test $R_1 \bar{\cup} R_2 \equiv R_1 \cup R_2$ is implemented in PIPS.

Translation from one store to another one The linear constraints defining a region often involve integer scalar variables from the program (e.g. ϕ_1==I). Their values, and thus the region, are relative to the current memory store. If we consider the statement `I = I + 1`, the value of `I` is not the same in the stores preceding and following the execution of the instruction. Thus, if the polyhedron of a region is ϕ_1==I before the execution of `I = I + 1`, it must be ϕ_1==I-1 afterwards.

To apply one of the preceding operators to two regions, they must be relative to the same store. Let $\mathcal{T}_{\sigma_1 \to \sigma_2}$ denote the transformation of a region relative to the store σ_1 into a region relative to the store σ_2.

This transformation is described in[3]. Very briefly, it consists in adding to the predicate of the region, the constraints of the transformer that abstracts the effects of the program between the two stores. The variables of the original store (σ_1) are then eliminated. The only variables that remain in the resulting polyhedron all refer to the store σ_2. Thus, two transformations, $\mathcal{T}_{\sigma_k \to \sigma_{k+1}}$ and $\mathcal{T}_{\sigma_{k+1} \to \sigma_k}$, correspond to the transformer T_k associated to statement S_k, depending on the variables that are eliminated.

For instance, let us assume that σ_1 is the store preceding the statement `I = I + 1`, σ_2 the store following it, and {ϕ_1==I} the predicate of a region relative to σ_1.

σ_1 {ϕ_1==I}
↓ `I = I + 1`
σ_2 {ϕ_1==I-1}

We first rename `I` into `I#init` in the predicate of the region, and add the transformer corresponding to the statement (`T(I) {I==I#init+1}`). This gives {ϕ_1==I#init, I==I#init+1}. We then eliminate `I#init`, because it refers to σ_1. We obtain {ϕ_1==I-1}, which is relative to σ_2.

The exactness of the operation depends on several factors, such as the combined characteristics of the transformer and the region, and the exactness of the variable elimination[2, 29]. When the operation is not exact, it leads to an over-approximation of the target region, which becomes a `MAY` region.

Merging over an iteration space The region corresponding to the body of a loop is a function of the value i of the loop index. During the propagation of regions, we shall need to merge regions corresponding to different, but successive, instances of the loop body, in order to get a summary over a particular iteration subspace ($\bigcup_{lb \leq i \leq ub} R(i)$).

By definition of the union of sets, this is strictly equivalent to eliminating the loop index from the region predicate, in which the description of the iteration subspace ($lb \leq i \leq ub$) has been added. However, the elimination of a variable from a region may lead to an over-approximation of the target region:

$$proj_i(R(i)_{lb \leq i \leq ub}) \; = \; \overline{\bigcup}_{lb \leq i \leq ub} R(i)$$

The operation is exact if the following conditions are met:

1. lb and ub are affine functions of the program integer scalar variables, for instance `do I = I1, I1+N-1`;

2. The elimination of i from $R(i)_{lb \leq i \leq ub}$ is exact according to the conditions of Ancourt or Pugh[2, 29][4].

The first condition ensures that the iteration space can be exactly described by a convex polyhedron over the program variables (here $lb \leq i \leq ub$)[5].

Constraining region predicates In order to have more information on ϕ variables, the constraints of the preconditions can be added to the predicate of the region. This is particularly useful when merging two regions.

> For instance, $\{\phi_1 \mid \phi_1$`==I`$\}$ $\bar{\cup}$ $\{\phi_1 \mid \phi_1$`==J`$\}$ is the whole space, i.e. an empty set of constraints. If the current precondition (e.g. $\{$`I==J`$\}$) is added to the original regions, the resulting region is $\{\phi_1 \mid \phi_1$`==I,I==J`$\}$ instead of $\{\phi_1 \mid \}$.

This operation increases the accuracy of the analysis, without modifying the definition of regions. Furthermore, since preconditions include some `IF` conditions, regions are powerful enough to disprove some interprocedurally conditional dependencies.

5 Intraprocedural Analyses

This section details the intraprocedural computation of READ, WRITE, IN and OUT regions for some of the main structures of the FORTRAN language: assignment, sequence of complex instructions and `DO` loop. The interprocedural propagation is described in Section 6.

5.1 READ and WRITE regions

Assignment The reference on the left hand side of the assignment is converted into a WRITE region, whereas on the right hand side, each reference is converted into an elementary READ region. These regions are exact if and only if the subscripts are affine functions of the program variables, for instance `A(2*I+3*J-1)`.

When several references to a particular array appear in the right hand side, the corresponding regions are systematically merged using $\bar{\cup}$ in order to obtain a summary.

> For instance, in Example 1, the elementary READ regions for the instruction `A(I) = A(I)+WORK(J,K)+WORK(J,K-1)` are:
>
> <A(ϕ_1)-R-EXACT-{ϕ_1==I}>
> <WORK(ϕ_1,ϕ_2)-R-EXACT-{ϕ_1==J, ϕ_2==K}>
> <WORK(ϕ_1,ϕ_2)-R-EXACT-{ϕ_1==J, ϕ_2==K-1}>
>
> By merging the two regions concerning the array `WORK`, we finally obtain:
>
> <A(ϕ_1)-R-EXACT-{ϕ_1==I}>
> <WORK(ϕ_1,ϕ_2)-R-EXACT-{ϕ_1==J, K-1<=ϕ_2<=K}>

[4] The elimination of variable v between the inequalities $av + A \leq 0$ and $-bv + B \leq 0$ (with $a \in \mathbb{N}^+$, $b \in \mathbb{N}^+$, $A = c + \sum_{i=1}^{\alpha} a_i v_i$, $B = d + \sum_{i=1}^{\beta} b_i v_i$, and $c, d, a_i, b_i \in \mathbb{Z}$), is exact if and only if $aB + bA + ab - a - b + 1 \leq 0$.

[5] Remember that the loop is normalized: the increment is equal to one.

Sequence of Instructions Our purpose is to compute the regions R_0 corresponding to the sequence S_1, S_2[6], that is to say a summary of all the read and write references occurring in S_1 and S_2.

R_1 and R_2, the READ and WRITE regions of S_1 and S_2, are supposed to be known. R_2 refers to the store σ_2 preceding the execution of S_2, while R_1 and R_0 refer to the store σ_1 preceding S_1 as well as the sequence S_1, S_2. Thus, we must first convert them into the same store (σ_1) before merging them:

$$R_0 = R_1 \bar{\cup} \mathcal{T}_{\sigma_2 \to \sigma_1}(R_2)$$

As an illustration, let us consider the body of the `I` loop in our example. We assume that we know the regions concerning the array `WORK` associated to the two inner loops:

```
C S1
C  <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N, φ2==K}>
      DO J = 1,N
      ...
C S2
      CALL INC1(K)
C S3
C  <WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N, φ2==K}>
C  <WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N, K-1<=φ2<=K}>
      DO J = 1,N
      ...
```

We cannot simply merge the regions associated to S_1 and S_3 to obtain the regions of the whole sequence, because the value of `K` is modified by S_2. They must first be converted into the store σ_2, by using $\mathcal{T}_{\sigma_3 \to \sigma_2}$: the transformer that abstracts the effects of the call to `INC1` is `T(K) {K==K#init+1}`; its constraint is added to the regions corresponding to S_3; then the variable `K`, which refers to the store immediately following S_2, is eliminated; and `K#init`, which represents the value of the variable `K` in σ_2, is renamed into `K`:

```
<WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N, φ2==K+1}>
<WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N, K<=φ2<=K+1}>
```

These regions are relative to the store preceding S_2. We should translate them to the store preceding S_1. However, since S_1 modifies no integer scalar variable, they are identical. Thus, it is legal to merge them with the regions corresponding to S_1, to obtain the regions for the sequence S_1, S_2, S_3:

```
<WORK(φ1,φ2)-W-EXACT-{1<=φ1<=N, K<=φ2<=K+1}>
<WORK(φ1,φ2)-R-EXACT-{1<=φ1<=N, K<=φ2<=K+1}>
```

DO loop

```
C   σ0
    DO I = lb, ub
C      σi
       S
    ENDDO
```

[6] S_2 can also be a sequence of instructions.

The purpose is to compute the regions corresponding to the loop and relative to σ_0, from the regions of its body S. These regions are not only functions of the value i of the loop index, but also of the variables v modified by S. Let $R(i, v)$ denote them.

First, we must get rid of the variables v in order to obtain regions that are functions of the sole loop index (and of course of variables that do not vary in the loop body). This is achieved by using $\mathcal{T}_{\sigma_i \to \sigma_0}$. This operator is based on the transformer of the loop, which gives the loop invariant when it is computable. We must then merge the resulting regions over the iteration space:

$$R_0 = \overline{\bigcup}_{lb \leq i \leq ub} \mathcal{T}_{\sigma_i \to \sigma_0}(R(i, v))$$

As an example, let us compute the READ regions of the array `WORK` for the loop `I` in Figure 1. As previously seen, the regions of the loop body are:

<WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N, K<=ϕ_2<=K+1}>

They are functions of the variable `K`, which is modified in the loop body by a call to `INC1`. To get rid of it, we must use the operator $\mathcal{T}_{\sigma_i \to \sigma_0}$: The transformer giving the loop invariant is `T(K) {K==K#init+I-1}` (`K#init` is here the value of `K` in the store preceding the loop); its constraint is added to the region, and `K` is eliminated; `K#init` is then renamed into `K`; and since all these steps are exact operations, we have:

<WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N, K+I-1<=ϕ_2<=K+I}>

To perform the union over the iteration space, the iteration space constraint (`{1<=I<=N}`) is added to the region, and then `I` is eliminated. This operation is exact because the lower and upper bounds are affine and the elimination of `I` is exact. We finally obtain:

<WORK(ϕ_1,ϕ_2)-R-EXACT-{1<=ϕ_1<=N, K<=ϕ_2<=K+N}>

5.2 IN and OUT Regions

READ and WRITE regions summarize the exact effects of statements and procedures upon array elements. They do not represent the flow of array element values, which are necessary to test the legality of many optimizations. For that purpose, we introduce two new types of regions: IN and OUT regions, which take array kills[30] into account, that is to say redefinitions of individual array elements.

IN regions contain the array elements, whose values are (`EXACT` region) or may be (`MAY` region) *imported* by the current piece of code. These are the elements that are read before being possibly redefined by another instruction of the same code fragment.

In Figure 1, the body of the second `J` loop reads the element `WORK(J,K)`, but does not imports its value because it is previously defined in the same iteration. On the contrary, the element `WORK(J,K-1)` is imported from the first `J` loop.

OUT regions corresponding to a piece of code contain the array elements that it defines, and that are (EXACT) or may be (MAY) used afterwards, in the continuation. These are the *live* or *exported* array elements.

In the program of Figure 1, the first J loop exports all the elements of the array WORK it defines towards the second J loop, whereas the elements of WORK defined in the latter are not exported towards the next iterations of the I loop.

In the remainder of this section, we limit ourselves to the intraprocedural computation of IN and OUT regions for an assignment, a sequence of instructions, or a loop.

5.2.1 IN Regions

Assignment The IN regions of an assignment are identical to the corresponding READ regions because the values of the referenced elements cannot come from the assignment itself, according to the FORTRAN standard.

Sequence of instructions We are now interested in the region IN_0 corresponding to the sequence of instructions S_1, S_2, and relative to the store σ_1 preceding the execution of S_1. It is the set of array elements imported by S_2 (IN_2) but not previously written by S_1 (W_1), merged with the set of array elements imported by S_1 (IN_1):

$$IN_0 = IN_1 \bar{\cup} (\mathcal{T}_{\sigma_2 \to \sigma_1}(IN_2) \ominus W_1)$$

As an illustration, let us consider the body of the second J loop in Figure 1. The READ and IN regions of its instructions concerning the array WORK are:

```
C S1
C  <WORK(φ1,φ2)-W-EXACT-{φ1==J, φ2==K}>
     WORK(J,K) = J*J - K*K
C S2
C  <WORK(φ1,φ2)-IN-EXACT-{φ1==J, K-1<=φ2<=K}>
     A(I) = A(I) + WORK(J,K) + WORK(J,K-1)
```

Since no scalar variable is modified in the sequence, we have :

$$\begin{aligned} IN_0 &= IN_1 \bar{\cup} (IN_2 \ominus W_1) \\ &= \varnothing \bar{\cup} (IN_2 \ominus W_1) \\ &= \texttt{<WORK}(\phi_1,\phi_2)\texttt{-IN-EXACT-}\{\phi_1\texttt{==J, }\phi_2\texttt{==K-1}\}\texttt{>} \end{aligned}$$

Finally, IN_0 contains the sole element WORK(J,K-1).

Loop We are now interested in the region IN_0 of a normalized `DO` loop, given the WRITE and IN regions of its body, respectively $W(i,v)$ and $IN(i,v)$; i is the value of the loop index, and v represents the variables modified by the loop body. Let σ_0 denote the store before the loop and σ_i the store before the iteration i.

We first get rid of the variables v using $\mathcal{T}_{\sigma_i \to \sigma_0}$. In order to simplify the next equation, we use the following notations:

$$W(i) = \mathcal{T}_{\sigma_i \to \sigma_0}(W(i,v))$$
$$IN(i) = \mathcal{T}_{\sigma_i \to \sigma_0}(IN(i,v))$$

The IN regions of a loop contain the array elements that are imported by each iteration ($IN(i)$) but not from the preceding iterations ($\overline{\bigcup}_{0 \le i' < i} W(i')$). The complete equation is then:

$$IN_0 = \overline{\bigcup}_{lb \le i \le ub} (\, IN(i) \, \ominus \, \overline{\bigcup}_{lb \le i' < i} W(i') \,)$$

The purpose of the following example is to compute the summary IN regions of the array `WORK` for the second `J` loop in Figure 1, given the WRITE and IN regions of its body:

$$\texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>}$$
$$\texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-EXACT-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K-1\}>}$$

Since no scalar variable is modified by the loop body, we can avoid the use of the operator $\mathcal{T}_{\sigma_i \to \sigma_0}$. We then compute the term $\overline{\bigcup}_{1 \le \texttt{J}' < \texttt{J}} W(\texttt{J}')$. We first add the iteration subspace constraint to the region:

$$\texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{}\phi_1\texttt{==J', }\phi_2\texttt{==K, 1<=J'<=J-1\}>}$$

By eliminating the loop index `J'`, we obtain the set of all the array elements written by at least one iteration preceding the iteration `J`:

$$\texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{1<=}\phi_1\texttt{<=J-1, }\phi_2\texttt{==K\}>}$$

These elements are then removed from the set of elements imported by the iteration `J`:

$$\begin{aligned} & \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-EXACT-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K-1\}>} \\ \ominus \; & \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{1<=}\phi_1\texttt{<=J-1, }\phi_2\texttt{==K\}>} \\ = \; & \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-EXACT-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K-1\}>} \end{aligned}$$

This last region represents the set of elements imported by the iteration `J` from the instructions preceding the loop. These regions are then merged over the whole iteration space ($1 \le \texttt{J} \le \texttt{N}$) to obtain the set of elements imported by at least one iteration, from the instructions preceding the loop:

$$\texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-EXACT-\{1<=}\phi_1\texttt{<=N, }\phi_2\texttt{==K-1\}>}$$

Hence, the loop imports all the values stored in the elements of array `WORK` such that ϕ_2`==K-1`.

5.2.2 *OUT Regions*

The OUT region of a statement is not defined *per se*, but depends on the future of the computation. For instance, the OUT region of S_1 in program S_1, S_2 is a function of S_1, S_2 as a whole, and of S_2. Thus, OUT regions are propagated in a top-down fashion along the call graph and hierarchical control flow graph of the program. Since I/O operations are part of the program, the OUT region of the main program, from which the other OUT regions are derived, is initialized to $\varnothing$.

Instructions of a sequence The region OUT_0 corresponding to the sequence S_1, S_2, and relative to the store σ_1 preceding S_1, is supposed to be known. The regions OUT_1 and OUT_2 corresponding to S_1 and S_2 are computed from OUT_0.

S_2 exports the elements that it writes (W_2) and that are exported by the whole sequence:

$$OUT_2 = W_2 \cap \mathcal{T}_{\sigma_1 \to \sigma_2}(OUT_0)$$

The elements exported by S_1 are those that it defines (W_1), and that are either exported by the whole sequence (OUT_0) but not by S_2 (OUT_2), or exported towards S_2, i.e. that are imported by S_2 (IN_2):

$$OUT_1 = W_1 \cap [\, (\, OUT_0 \ominus \mathcal{T}_{\sigma_2 \to \sigma_1}(OUT_2)\,) \cup \mathcal{T}_{\sigma_2 \to \sigma_1}(IN_2)\,]$$

Let us consider as an illustration the body of the second `J` loop, in Figure 1. Its WRITE and IN regions for the array `WORK` are:

```
C S1
C  <WORK(φ1,φ2)-W-EXACT-{φ1==J, φ2==K}>
     WORK(J,K) = J*J - K*K
C S2
C  <WORK(φ1,φ2)-IN-EXACT-{φ1==J, K-1<=φ2<=K}>
     A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
```

Since no integer scalar variable is modified by the loop body, $\mathcal{T}_{\sigma_1 \to \sigma_2}$ and $\mathcal{T}_{\sigma_2 \to \sigma_1}$ are identity. Moreover, we assume that $OUT_0 = \varnothing$. The derivation is:

$$\begin{aligned}
OUT_2 =& W_2 \cap OUT_0 = \varnothing \\
OUT_1 =& W_1 \cap [(OUT_0 \ominus OUT_2) \cup IN_2] \\
=& W_1 \cap IN_2 \\
=& \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>} \\
& \cap \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-EXACT-\{}\phi_1\texttt{==J, K-1<=}\phi_2\texttt{<=K\}>} \\
=& \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-EXACT-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>}
\end{aligned}$$

S_1 exports the element it defines towards S_2, which exports no element of `WORK`.

Loop body The goal is to compute the OUT regions of the loop body ($OUT(i)$ if i is the value of the loop index) from the regions of the whole loop (OUT_0). An array element can be exported by the iteration i for two reasons:

1. Either it is written by the iteration i ($W(i)$), and exported towards the continuation of the loop (i.e. it belongs to OUT_0); but it must not be re-defined by any subsequent iteration; in other words, it must not belong to the set of array elements defined by the iterations i' such that $i < i' \leq ub$: $\overline{\bigcup}_{i<i'\leq ub}(W(i'))$; thus, it belongs to the region defined by:

$$(W(i) \cap \mathcal{T}_{\sigma_0 \to \sigma_i}(OUT_0)) \ominus \overline{\bigcup}_{i<i'\leq ub}(W(i'))$$

2. Or, it is written by the iteration i ($W(i)$), and directly used in a subsequent iteration i'; *directly* means that it must not be defined by an iteration i'' between i and i':

$$W(i) \cap \overline{\bigcup}_{i<i'\leq ub}[IN(i') \ominus \overline{\bigcup}_{i<i''<i'}(W(i''))]$$

And finally, the complete equation is:

$$OUT(i) = \{(W(i) \cap \mathcal{T}_{\sigma_0 \to \sigma_i}(OUT_0)) \ominus \overline{\bigcup}_{i<i'\leq ub}(W(i'))\} \bar{\cup} \{W(i) \cap \overline{\bigcup}_{i<i'\leq ub}[IN(i') \ominus \overline{\bigcup}_{i<i''<i'}(W(i''))]\}$$

Let us take an example to illustrate some features of the previous equation. We consider the `I` loop in the program of figure 1. The goal is to compute the OUT regions concerning the array `A` for the loop body. We assume that its WRITE and IN regions are already available:

`<A(`ϕ_1`)-W-EXACT-{`ϕ_1`==I}>`
`<A(`ϕ_1`)-IN-EXACT-{`ϕ_1`==I}>`

and that the OUT regions of the whole loop (OUT_0) are:

`<A(`ϕ_1`)-OUT-EXACT-{1<=`ϕ_1`<=N}>`

$\mathcal{T}_{\sigma_0 \to \sigma_i}(OUT_0)$ is first calculated: the constraints of the loop transformer, `T(K){K==K#INIT+I-1}`, are added to the polyhedron of the region, and `K#INIT` is eliminated:

`<A(`ϕ_1`)-OUT-EXACT-{1<=`ϕ_1`<=N}>`

Then, we compute $W(i) \cap \mathcal{T}_{\sigma_0 \to \sigma_i}(OUT_0)$:

`<A(`ϕ_1`)-OUT-EXACT-{`ϕ_1`==I, 1<=`ϕ_1`<=N}>`

and $\overline{\bigcup}_{i<i'\leq ub}(W(i'))$ $(= proj_{i'}(W(i')_{i<i'\leq ub}))$:

$W(i')_{i<i'\leq ub}$ =`<A(`ϕ_1`)-W-EXACT-{`ϕ_1`==I', I+1<=I'<=N}>`
$proj_{i'}(W(i')_{i<i'\leq ub})$ =`<A(`ϕ_1`)-W-EXACT-{I+1<=`ϕ_1`<=N}>`

Finally, the first part of the equation gives the region:

$$\texttt{<A(}\phi_1\texttt{)-OUT-EXACT-\{}\phi_1\texttt{==I, 1<=}\phi_1\texttt{<=N\}>}$$

For the second part of the equation, we successively have:

$$\begin{aligned}
\overline{\bigcup}_{i<i''<i'}(W(i'')) &= \texttt{<A(}\phi_1\texttt{)-W-EXACT-\{I+1<=}\phi_1\texttt{<=I'-1\}>} \\
IN(i') \ominus \overline{\bigcup}_{i<i''<i'}(W(i'')) &= \texttt{<A(}\phi_1\texttt{)-IN-EXACT-\{}\phi_1\texttt{==I'\}>} \\
&\quad \ominus \texttt{<A(}\phi_1\texttt{)-W-EXACT-\{I+1<=}\phi_1\texttt{<=I'-1\}>} \\
&= \texttt{<A(}\phi_1\texttt{)-IN-EXACT-\{}\phi_1\texttt{==I'\}>}
\end{aligned}$$

and,

$$\begin{aligned}
\overline{\bigcup}_{i<i'\leq n}[\ldots] &= \texttt{<A(}\phi_1\texttt{)-IN-EXACT-\{I+1<=}\phi_1\texttt{<=N\}>} \\
W(i) \cap \overline{\bigcup}_{i<i'\leq n}[\ldots] &= \texttt{<A(}\phi_1\texttt{)-W-EXACT-\{}\phi_1\texttt{==I\}>} \\
&\quad \cap \texttt{<A(}\phi_1\texttt{)-IN-EXACT-\{I+1<=}\phi_1\texttt{<=N\}>} \\
&= \varnothing
\end{aligned}$$

Thus, the iteration i exports no element of `A` towards the subsequent iterations. And finally, for the whole equation, and for each iteration i, the region is:

$$\texttt{<A(}\phi_1\texttt{)-OUT-EXACT-\{}\phi_1\texttt{==I, 1<=}\phi_1\texttt{<=N\}>}$$

The complete IN and OUT regions of our example are given in Figure 4. They show that the body of the `I` loop imports and exports no element of `WORK`, which can be privatized by PIPS after induction variable substitution (see Figure 5).

6 Interprocedural Analyses

The intraprocedural computation of array regions has been described in the previous section. We now focus on the interprocedural part of array region analyses. The first subsection is devoted to the propagation on the call graph, while the second extensively describes the translation of array regions from the source procedure name space to the target procedure name space.

6.1 Propagation on the call graph

The interprocedural propagation of READ, WRITE, and IN regions is a backward (or bottom-up) analysis. At each call site the summary regions of the called subroutine are translated from the callee's name space into the caller's name space, using the relations between actual and formal parameters, and between the declarations of global variables in both routines. This is illustrated in the leftmost picture of Figure 6.

On the contrary, the interprocedural propagation of OUT regions is a forward (or top-down) analysis. The regions of all the call sites are first translated from the callers' name space into the callee's name space, and are then merged to form a unique summary[7] (see the rightmost picture in Figure 6).

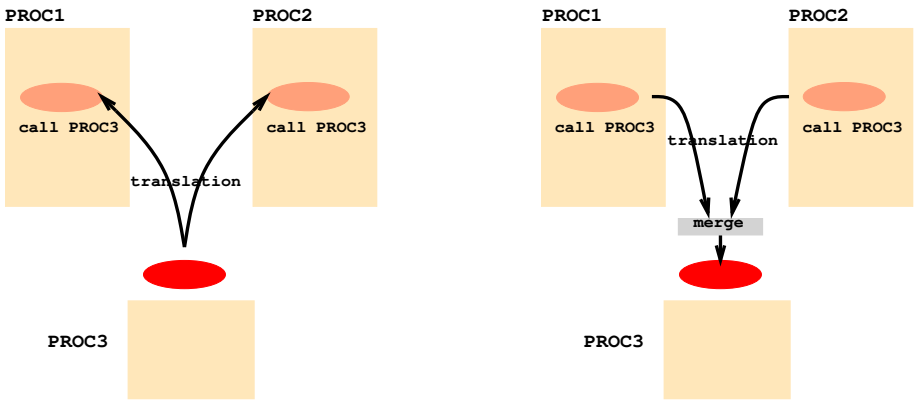
[7] The OUT regions of the main routine are initialized to $\varnothing$ (see Section 5.2.2).

```
      K = FOO()

C  <A(φ1)-IN-MUST-{1<=φ1<=N}>
      DO I = 1, N

C  loop body:
C  <A(φ1)-IN-MUST-{φ1==I, 1<=φ1<=N}>

C  <WORK(φ1,φ2)-OUT-MUST-{φ2==K, 1<=φ1<=N}>
         DO J = 1, N
C  <WORK(φ1,φ2)-OUT-MUST-{φ1==J, φ2==K, 1<=φ1<=N}>
            WORK(J,K) = J+K
         ENDDO
         CALL INC1(K)

C  <A(φ1)-IN-MUST-{I==φ1}>
C  <WORK(φ1,φ2)-IN-MUST-{φ2==K-1, 1<=φ1<=N}>
         DO J = 1, N

C  <WORK(φ1,φ2)-OUT-MUST-{φ1==J, φ2==K}>
            WORK(J,K) = J*J-K*K
C  <WORK(φ1,φ2)-IN-MUST-{φ1==J, K-1<=φ2<=K}>
C  <A(φ1)-IN-MUST-{φ1==I}>
C  <A(φ1)-OUT-MUST-{φ1==I, 1<=J<=N-1}>
            A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
         ENDDO
      ENDDO
```

Fig. 4. IN and OUT regions.

```
   K0 = FOO()
   DOALL I = 1, N
   PRIVATE WORK,J,K
      K = K0+I-1
      DOALL J = 1, N
        WORK(J,K) = J+K
      ENDDO
      CALL INC1(K)
      DOALL J = 1, N
        WORK(J,K) = J*J-K*K
      ENDDO
      DO J = 1, N
         A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
      ENDDO
   ENDDO
```

Fig. 5. Parallel version.



a) Backward propagation: READ, WRITE, and IN regions.

b) Forward propagation: OUT regions.

Fig. 6. Interprocedural propagation of array regions.

6.2 Array region translation

This section describes the translation part of the interprocedural propagation. Because the source and target variables may not have the same declaration (*array reshaping*), this operation is not straightforward.

By examining the Perfect Club benchmarks[5], we found it necessary to handle several non-exclusive cases:

1. Array reshaping due to different dimension declarations.
2. Offsets between the first elements of the source and target arrays due to parameter passing (`CALL F(A(1,J))` for instance);
3. Offsets due to different `COMMON` declarations in the caller and the callee (e.g. in the program `TRFD`, the common `TR2PRT` is not similarly declared in routines `TRFPRT` and `TRFOUT`).
4. Target and source variables of different types (e.g.in the program `OCEAN`).

The method described in this section tackles these four points. It is based on the fact that two corresponding elements of the source and target arrays must have the same subscript values[8], up to the offset between their first element. This is described in section 6.2.2.

However, the resulting *translation system* may contain non-linear terms, and it hides the trivial relations existing between the ϕ variables of both arrays. Hence, we propose in section 6.2.3 an algorithm that first tries to discover these trivial relations before using the subscript values. It results in a *simplified translation system*.

[8] The subscript value of an array elements is its *rank* in the array, array elements being held in column order.[1]

6.2.1 Notations

In the remainder of this section, we use the following notations:

	source	$\longmapsto$	*target*
array	A		B
dimension	α		β
lower bounds	$l_{a_1}, \ldots, l_{a_\alpha}$		$l_{b_1}, \ldots, l_{b_\beta}$
upper bounds	$u_{a_1}, \ldots, u_{a_\alpha}$		$u_{b_1}, \ldots, u_{b_\beta}$
size of elements[9]	s_a		s_b
region parameters	$\phi_1, \ldots, \phi_\alpha$		$\psi_1, \ldots, \psi_\beta$

The subscript values of $A(\phi_1, \ldots, \phi_\alpha)$ and $B(\psi_1, \ldots, \psi_\beta)$ are thus[10]:

$$subscript_value(A(\phi_1, \ldots, \phi_\alpha)) = \sum_{i=1}^{\alpha} [(\phi_i - l_{a_i}) \prod_{j=1}^{i-1} (u_{a_j} - l_{a_j} + 1)]$$

$$subscript_value(B(\psi_1, \ldots, \psi_\beta)) = \sum_{i=1}^{\beta} [(\psi_i - l_{b_i}) \prod_{j=1}^{i-1} (u_{b_j} - l_{b_j} + 1)]$$

Another necessary information is the offset of the first element of A from the first element of B in the memory layout. This information is provided differently, depending on the type of aliasing between A and B:

source parameter $\mapsto$ *target parameter*	*offset*
formal $\mapsto$ *actual*	reference at call site: $B(o_{b_1}, \ldots, o_{b_\beta})$ $offset = s_b \times subscript_value(B(o_{b_1}, \ldots, o_{b_\beta}))$
actual $\mapsto$ *formal*	reference at call site: $A(o_{a_1}, \ldots, o_{a_\alpha})$ $offset = -s_a \times subscript_value(A(o_{a_1}, \ldots, o_{a_\alpha}))$
global $\mapsto$ *global*	numerical offset difference between the offset of A in the declaration of the common in the source subroutine, and the offset of B in the declaration of the common in the target subroutine.

As an illustration, let us consider the contrived program in Figure 7, which contains all the difficulties encountered in real life programs. The purpose is to find the READ and WRITE regions of the call site, from the summary regions of procedure `BAR`. The translation coefficients are:

`R` $\mapsto$ *C:* $A =$ `R`, $B =$ `C`; $\alpha = 2$, $\beta = 3$; $l_{a_1} = l_{a_2} = 1$, $l_{b_1} = l_{b_2} = lb_3 = 1$; $u_{a_1} =$ `n1`, $u_{a_2} =$ `n2`; $u_{b_1} =$ `n`, $u_{b_2} = 10$, $u_{b_3} = 20$; $s_a = 4$, $s_b = 8$; $offset = 0$;

[9] Unit: the size of the smallest accessible amount of memory (usually one byte).

[10] With the convention that $\prod_{k=k_1}^{k_2} = 1$ when $k_2 < k_1$.

```
                                        C  <D2(φ1,φ2)-W-EXACT-{1<=φ1<=10, 1<=φ2<=9}>
                                        C  <D1(φ1)-W-EXACT-{1<=φ1<=10}>
                                        C  <R(φ1,φ2)-W-EXACT-{1<=φ1<=N1, 1<=φ2<=N2}>
subroutine FOO(C,n)                        subroutine BAR(R,n1,n2)
complex C(n,10,20),D                       real R(n1,n2)
common D(5,10)                             common D1(10), D2(10,9)
call BAR(C,2n,100)                         ...
end                                        end
```

Fig. 7. Interprocedural translation: example.

$D1 \mapsto D$: $A = \mathtt{D1}$, $B = \mathtt{D}$; $\alpha = 1$, $\beta = 2$; $l_{a_1} = 1$, $l_{b_1} = l_{b_2} = 1$; $u_{a_1} = 10$; $u_{b_1} = 5$, $u_{b_2} = 10$; $s_a = 4$, $s_b = 8$; $offset = 0$;
$D2 \mapsto D$: $A = \mathtt{D2}$, $B = \mathtt{D}$; $\alpha = 2$, $\beta = 2$; $l_{a_1} = l_{a_2} = 1$, $l_{b_1} = l_{b_2} = 1$; $u_{a_1} = 10$, $u_{a_2} = 9$; $u_{b_1} = 5$, $u_{b_2} = 10$; $s_a = 4$, $s_b = 8$; $offset = 40$.

6.2.2 General translation system

With the previous notations, the region parameters of the element $B(\psi_1, \ldots, \psi_\beta)$ corresponding to the source element $A(\phi_1, \ldots, \phi_\alpha)$ must verify the following system:

$$\exists\, \delta_a, \delta_b / \begin{cases} s_a \times subscript_value(A(\phi_1, \ldots, \phi_\alpha)) + \delta_a + offset \\ \quad = s_b \times subscript_value(B(\psi_1, \ldots, \psi_\beta)) + \delta_b \\ 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{cases} \qquad (S)$$

δ variables are used to describe the corresponding elementary memory cells inside two associated array elements, as shown in Figure 8.

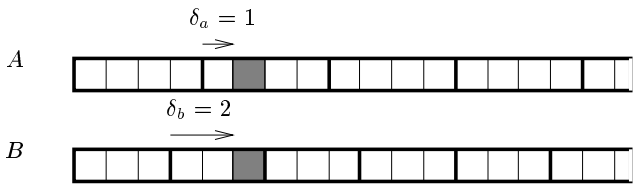


Fig. 8. Meaning of δvariables.

For our example, the following systems would be built:

$R \mapsto C$:

$$\begin{cases} 4[(\phi_1 - 1) + \mathtt{n1}(\phi_2 - 1)] + \delta_a = \\ \quad 8[(\psi_1 - 1) + \mathtt{n}(\psi_2 - 1) + 10\mathtt{n}(\psi_3 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8,\ \mathtt{n1} = 2\mathtt{n} \end{cases}$$

D1 $\mapsto$ *D:*

$$\begin{cases} 4(\phi_1 - 1) + \delta_a = 8[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8 \end{cases}$$

D2 $\mapsto$ *D:*

$$\begin{cases} 4[(\phi_1 - 1) + 10(\phi_2 - 1)] + \delta_a + 40 = 8[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_a < 4,\ 0 \leq \delta_b < 8 \end{cases}$$

Using S as the translation system has several drawbacks:

1. in the *formal* $\leftrightarrow$ *actual* cases, S is generally non-linear (it is the case in our first example);
2. in order to keep a convex representation, δ variables must be eliminated; this operation may be inexact, leading to an over-approximation;
3. even in favorable cases, the equation in system S is rather complex, and hides the trivial relations existing between ϕ and ψ variables, such as $\phi_1 = \psi_1$; this makes the subsequent analyses unnecessarily complex, and is not acceptable in an interactive environment.

In the following section, we describe a method that alleviates these three problems.

6.2.3 Simplified translation system

Elimination of unnecessary δ variables

Theorem 1. *If s_b divides s_a and offset, then S is equivalent to the following system*[11]*:*

$$\exists\, \delta'_a / \begin{cases} s'_a \times \mathit{subscript_value}(A(\phi_1, \ldots, \phi_\alpha)) + \delta'_a + \dfrac{\mathit{offset}}{s_b} \\ \quad = \mathit{subscript_value}(B(\psi_1, \ldots, \psi_\beta)) \\ 0 \leq \delta'_a < s'_a \\ s'_a = \dfrac{s_a}{s_b} \end{cases}$$

Note.

1. In the *formal* $\leftrightarrow$ *actual* cases, s_b divides $s_a \Rightarrow s_b$ divides *offset*.
2. In fact, we just replace s_a by $\frac{s_a}{s_b}$, s_b by 1, *offset* by $\frac{\mathit{offset}}{s_b}$ and use S without δ_b.

In our working example, since s_a divides s_b and *offset* in all three cases, the translation systems become:

R $\mapsto$ *C:*

$$\begin{cases} (\phi_1 - 1) + \mathtt{n1}(\phi_2 - 1) = \\ \quad 2[(\psi_1 - 1) + \mathtt{n}(\psi_2 - 1) + 10\mathtt{n}(\psi_3 - 1)] + \delta_b \\ 0 \leq \delta_b < 2,\ \mathtt{n1} = 2\mathtt{n} \end{cases}$$

[11] Of course, there is a similar system if s_a divides s_b and *offset*.

`D1` $\mapsto$ `D:`
$$\begin{cases} \phi_1 - 1 = 2[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_b < 2 \end{cases}$$
`D2` $\mapsto$ `D:`
$$\begin{cases} (\phi_1 - 1) + 10(\phi_2 - 1) + 10 = 2[(\psi_1 - 1) + 5(\psi_2 - 1)] + \delta_b \\ 0 \leq \delta_b < 2 \end{cases}$$

Decreasing the degree of (S)

Definition 2. *(Similar dimensions)*
A dimension d ($d \leq min(\alpha, \beta)$) is said to be similar for arrays A and B if the following three conditions are met:

1. *Condition for the offset:*
   There must be no offset between the first element of B and the first element of A on dimension d:

$formal \mapsto actual$	$\forall\, i / 1 \leq i \leq d,\ o_{b_i} = l_{b_i}$
$actual \mapsto formal$	$\forall\, i / 1 \leq i \leq d,\ o_{a_i} = l_{a_i}$
$global \mapsto global$	$\lvert offset \rvert \bmod s_a \prod_{i=1}^{d}(u_{a_i} - l_{a_i} + 1) = 0$
	$\wedge \lvert offset \rvert \bmod s_b \prod_{i=1}^{d}(u_{b_i} - l_{b_i} + 1) = 0$

2. *Condition for the first dimension:*
   The lengths in bytes of the first dimensions of A and B are equal:
   $$s_a(u_{a_d} - l_{a_d} + 1) = s_b(u_{b_d} - l_{b_d} + 1)$$
   This means that the first dimension entirely compensates the difference between s_a and s_b. This is why s_a and s_b are not used in the next condition.
3. *Condition for the next dimensions ($2 \leq d \leq min(\alpha, \beta)$):*
   Assuming that the previous dimensions are similar, the lengths of the d-th dimensions of A and B must be equal:
   $$u_{a_d} - l_{a_d} = u_{b_d} - l_{b_d}$$
   This is not necessary if $d = \alpha = \beta$.

This definition only takes into account dimensions of identical ranks. The general case would try to discover minimal sets of globally similar dimensions.

> For instance if the dimensions of A and B are $A(l, m, n)$ and $B(m, l, n)$, the global lengths of dimensions 1 and 2 are similar (dimensions 1 and 2 are globally similar); as a consequence, the third dimension is similar.

But the complexity of the algorithm for discovering these sets would be too high compared to the expected gain, especially in real life programs.

Notations. We now use the following notations for $k \in [2..min(\alpha, \beta)]$:

k_subscript_value:

$$k_subscript_value(A(\phi_1,\ldots,\phi_\alpha)) = \sum_{i=k}^{\alpha}[(\phi_i - l_{a_i})\prod_{j=k}^{i-1}(u_{a_j} - l_{a_j} + 1)]$$

It is the rank of the array element $A(\phi_1,\ldots,\phi_\alpha)$ from the element $A(\phi_1,\ldots,\phi_{k-1},l_{a_k},\ldots,l_{a_\alpha})$, i.e. from the first element of the k-th dimension.

k_offset:

It is the offset relative to the k-th dimension:

$formal \mapsto actual$	$k_subscript_value(B(o_{b_1},\ldots,o_{b_\beta}))$
$actual \mapsto formal$	$-k_subscript_value(A(o_{a_1},\ldots,o_{a_\alpha}))$
$global \mapsto global$	$\left\lfloor \frac{offset}{sa\prod_{i=1}^{k}(u_{a_i}-l_{a_i}+1)} \right\rfloor$

Theorem 3. *If dimensions 1 to $d-1$ ($1 \leq d-1 \leq min(\alpha,\beta)$) are similar, then S is equivalent to:*

$$\exists\, \delta_a, \delta_b / \begin{cases} s_a(\phi_1 - l_{a_1}) + \delta_a = s_b(\psi_1 - l_{b_1}) + \delta_b \\ \forall\, i \in [2..d-1],\ \ \phi_i - l_{a_i} = \psi_i - l_{b_i} \\ d_subscript_value(A(\phi_1,\ldots,\phi_\alpha)) + d_offset = \\ \quad d_subscript_value(B(\psi_1,\ldots,\psi_\beta))^{12} \\ 0 \leq \delta_a < s_a \\ 0 \leq \delta_b < s_b \end{cases} \qquad (S_d)$$

In our working example, the translation systems finally become:

`R` $\mapsto$ `C`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ \phi_2 - 1 = (\psi_2 - 1) + 10(\psi_3 - 1) \\ 0 \leq \delta_b < 2 \end{cases}$$

Notice that the system now only contains linear equations.

`D1` $\mapsto$ `D`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ (\psi_2 - 1) = 0 \\ 0 \leq \delta_b < 2 \end{cases}$$

There are now only very simple relations between ϕ and ψ variables. In particular, it becomes obvious that $\psi_2 = 1$, which was hidden in the original system.

`D2` $\mapsto$ `D`:

$$\begin{cases} \phi_1 - 1 = 2(\psi_1 - 1) + \delta_b \\ (\phi_2 - 1) + 1 = (\psi_2 - 1) \\ 0 \leq \delta_b < 2 \end{cases}$$

[12] In the formal $\mapsto$ actual case, if $d = min(\alpha,\beta) = \alpha$, this equation can be replaced by $\forall\, i \in [d..\beta],\ \psi_i = o_{b_i}$.

Notice how the offset for the whole problem has been turned into an offset for the sole second dimension (the term +1 in the second equation).

And at last, the translation algorithm is the following:

Algorithm.

1. input: a region R_A corresponding to the array A
2. $R_B = R_A$
3. $d =$ *number_of_similar_dimensions*$(A, B) + 1$
4. if $d = 1$ then
5. *translation_system* $= S$
6. else
7. *translation_system* $= S_d$
8. endif
9. add *translation_system* to R_B
10. eliminate δ variables
11. eliminate ϕ variables
12. rename ψ variables into ϕ variables
13. translate R_B's polyhedron into the target routine's name space
14. for all $i \in [1..\beta]$ add $l_{b_i} \leq \phi_i \leq u_{b_i}$ to R_B
15. output: R_B

At each step, the exactness of the current operation is checked. At Step 3, if an intermediate expression used to check the similarity is not linear, the current dimension is declared as non-similar, and the next dimensions are not considered. At Steps 5 and 7, if a constraint cannot be built because of a non-linear term, it is not used (this leads to an over-approximation of the solution set), and the translation is declared inexact. At Steps 10 and 11, the exactness of the variable elimination is verified with the usual conditions[2, 29].

Step 13 is performed using the relations between formal and actual parameters, and between the declarations of global variables in the source and target routines (this gives a *translation context system*). The variables belonging to the name space of the source routine are then eliminated. The exactness of this operation depends on the combined characteristics of the *translation context system* and R, and the exactness of the variable elimination[2, 29].

The last step is particularly useful in case of a partial matching between A and B, which is the case when A and B belong to a COMMON that is not similarly declared in the source and target routine.

For the example of Figure 7, the resulting regions are all exact:

<C(ϕ_1,ϕ_2,ϕ_3)-W-EXACT-{1<=ϕ_1<=N,1<=ϕ_2<=10,1<=ϕ_3<=20,ϕ_2+10ϕ_3<=110}>
<D(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=5, 2<=ϕ_2<=10}>
<D(ϕ_1,ϕ_2)-W-EXACT-{1<=ϕ_1<=5, ϕ_2==1}>

7 Related Work

The previous work closest to ours are those of Triolet[33], Tang[31], Hall[18], Li[27, 17] and Leservot[23], and the works by Burke and Cytron[8] and Maslov[26] for the interprocedural translation.

Many other less recent studies[9, 19, 4] have addressed the problem of the interprocedural propagation of array element sets. But they did not provide sufficient symbolic analyses, and did not tackle array reshaping.

Triolet[33] Array regions were originally defined by Triolet as *over-approximations* of the effects of statements of procedures upon sets of array elements (`MAY` READ and WRITE regions). We have extended his work to introduce the notion of exactness, and IN and OUT regions to represent the flow of array elements.

In his thesis[32], Triolet addressed the problem of interprocedural translation in a very limited way: no array reshaping, except when due to an offset in the actual parameter to represent a column in a matrix for instance; and the `COMMON`s in the caller and callee had to be similarly declared.

Tang[31] Tang summarizes multiple array references in the form of an integer programming problem. It provides exact solutions, but the source language is very restricted, and array reshaping is only handled in very simple cases (sub-arrays, as Triolet[32]).

Hall* et al.*[18] Fiat/Suif includes an intra- and inter-procedural framework for the analysis of array variables. Under- and over-approximations of array elements sets are represented by lists of polyhedra. The problem of exactness is not considered. However the list representation is more precise than ours, and the exactness of our regions would certainly benefit from it; but the cost, both in memory use and computation time, would certainly be more important.

Different types of regions are available in Fiat/Suif. The *Read* and $Write$ sets are similar to our READ and WRITE regions. However, the $ExposedRead$ sets contain the array elements which are used in the continuation of the whole program before being defined, while our IN regions are restricted to the current level in the HCFG. There are no equivalent for our OUT regions, which are (among other applications) useful for the interprocedural resolution of the *copy-out* problem in array privatization[24].

For the interprocedural translation, they have adopted a method basically similar to ours. However, in Fiat/Suif, similar dimensions are taken into account only when the system is not linear; and in this case, they do not try to build a system similar to S_d (see Page 23), possibly missing a linear translation system. Moreover, they do not handle global $\mapsto$ global translation when the `COMMON` to which the source and target arrays belong, does not have the same data layout in the caller and callee.

Li* et al.*[27, 17] In the Panorama compiler, the representation of array element sets is a list of RSDs[9] with bounds and step, guarded by predicates derived from `IF` conditions. Since our regions also include some `IF` conditions, the advantages of this representation over ours (except the use of lists) is unclear.

They also have different types of array element sets. MOD sets are similar to WRITE regions, and UE sets to IN regions; this is due to the fact that their analyses rely on a hierarchical control flow graph inspired from PIPS' HCFG[27]. But as in Fiat/Suif, there is no equivalent for our OUT regions.

The previous sets are exact sets, unless they contain an unknown component. Our regions should be more accurate, because we can keep information about all the ϕ variables, even in case of a `MAY` region.

Leservot[23] Leservot has extended Feautrier's array data flow analysis[16] to handle static control programs with procedure calls. To preserve the *a priori* determinism of the analysis, no partial association is allowed at procedure boundaries (i.e. the source and target arrays have the same type), and only very simple array reshapes are handled (the same cases as in[32] and[31]).

For each procedure, this method computes *in-going* effects, which bear some resemblance with IN regions, and *out-going* effects, which are somewhat similar to downward exposed writes, and are thus different from OUT regions.

Burke and Cytron[8] They alleviate the memory disambiguation problem by linearizing all array accesses when possible. This is equivalent to using the system S in our method. However, we have seen that this may lead to non linear expressions, that prevent further dependence testing for instance. On the contrary, our method avoids linearization whenever possible by detecting similar dimensions, and partially linearizing the remaining dimensions if possible and necessary. This approach eliminates the linearization versus subscript-by-subscript problem as formulated by Burke and Cytron.

Maslov[26] Maslov describes a very general method for simplifying systems containing polynomial constraints. This is the case of the general translation system presented in Section 6.2.3.

We think that most cases that arise in real life programs and that can be solved using Maslov's method can also be solved by our algorithm, thus avoiding the cost of a more general method; for instance, the translation from `A(N,M,L)` to `B(N,M,L)` yields the equation $\psi_1 + \mathtt{N}\psi_2 + \mathtt{NM}\psi_3 = \phi_1 + \mathtt{N}\phi_2 + \mathtt{NM}\phi_3$ which he gives as an example; we solve it by simply verifying that all three dimensions are similar.

8 Conclusion

Obviously, a lot of efforts have been spent over the last ten years to summarize memory effects on array elements. Time and space complexity, accuracy and usefulness are the usual issues. In PIPS, we have chosen to use convexity to reduce space complexity. We define several types of summaries.

READ and WRITE array regions represent the exact effects of statements and procedures upon array elements whenever possible. Whereas the regions initially defined by Triolet[33] are over-approximations of the effects of procedures. READ and WRITE regions are used by Coelho[10] to efficiently compile HPF.

Since READ and WRITE regions cannot be used to compute the flow of array elements, we have introduced two new types of exact array region. IN and OUT regions represent the sets of array elements that are imported or exported by the corresponding code fragment. IN regions contain the locally upward exposed read elements, and are thus different from the usual upward-exposed read references. IN and OUT regions are already used in PIPS for the privatization of array sections[3, 13] even when there are procedure calls.

We also provide a general linear framework for the interprocedural propagation of regions, regardless of their type. It handles array reshapes, even in `COMMON`s that do not have the same data layout, and when arrays do not have the same type. It is different from the other approaches because it systematically tries to discover similar dimensions, and uses linearization techniques only for the dimensions that are not similar.

The current implementation in PIPS covers all the intraprocedural structures of the FORTRAN language, along with the interprocedural propagation. A first series of experiments carried on the Perfect Club benchmarks shows the practicality of the analysis in terms of time and space, in spite of the well-known exponential complexity of operators on polyhedra.

More experiments are needed to determine if the representation of IN and OUT regions in polyhedral form is precise enough in general to perform optimizations such as array privatization, generation of communications in distributed memory machines, or compile-time optimization of cache behavior in hierarchical memory machines. Other representations are being considered, such as finite unions of polyhedra, and intersection of polyhedra and lattices.

Acknowledgments

We are very thankful to Corinne Ancourt, Fabien Coelho, Pierre Jouvelot and William Pugh for their careful reading of previous versions and helpful comments. We also wish to give special thanks to the referees for the improvements they suggested.

References

1. American National Standard Institute. *Programming Language FORTRAN, ANSI X3.9-1978, ISO 1539-1980*, 1983.
2. Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.
3. Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux (Exact array region analyses and array privatization). Master's thesis, Université Paris VI, France, September 1994. Available via `http://www.cri.ensmp.fr/~creusil`.

4. V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *International Conference on Programming Language Design and Implementation*, pages 41–53, June 1989.
5. M. Berry, D. Chen, P. Koss, D. Kuck, V. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT Club benchmarks : Effective performance evaluation of supercomputers. Technical Report CSRD-827, CSRD, University of Illinois, May 1989.
6. W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.
7. Thomas Brandes. The importance of direct dependences for automatic parallelization. In *International Conference on Supercomputing*, pages 407–417, July 1988.
8. Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 21(7):162–175, July 1986.
9. D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.
10. Fabien Coelho. Compilation of I/O communications for HPF. In *Frontiers'95*, pages 102–109, February 1995. Available via `http://www.cri.ensmp.fr/~coelho`.
11. Fabien Coelho and Corinne Ancourt. Optimal compilation of HPF remappings. Technical Report A-277-CRI, CRI, École des Mines de Paris, October 1995. To appear in JPDC in 1996.
12. Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.
13. Béatrice Creusillet. Array regions for interprocedural parallelization and array privatization. Report A-279, CRI, École des Mines de Paris, November 1995. Available at `http://www.cri.ensmp.fr/~creusil`.
14. Béatrice Creusillet. IN and OUT array region analyses. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 233–246, June 1995.
15. Béatrice Creusillet and François Irigoin. Interprocedural array regions analyses. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer-Verlag, August 1995.
16. Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, September 1991.
17. Jungie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, December 1995.
18. Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer-Verlag, August 1995.
19. Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.
20. François Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, pages 333–350, September 1992.

21. François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.
22. Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software : Practice and Experience*, 24(10):871–886, October 1994.
23. Arnauld Leservot. *Analyses interprocédurales du flot des données*. PhD thesis, Université Paris VI, March 1996.
24. Zhiyuan Li. Array privatization for parallel execution of loops. In *International Conference on Supercomputing*, pages 313–322, July 1992.
25. Vadim Maslov. Lazy array data-flow analysis. In *Symposium on Principles of Programming Languages*, pages 311–325, January 1994.
26. Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. Technical Report CS-TR-3109.1, University of Maryland, College Park, February 1994.
27. Trung Nguyen, Jungie Gu, and Zhiyuan Li. An interprocedural parallelizing compiler and its support for memory hierarchy research. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 96–110. Springer-Verlag, August 1995.
28. Michael Paleczny, Ken Kennedy, and Charles Koelbel. Compiler support for out-of-core arrays on parallel machines. In *Frontiers'95*, pages 110–118, February 1995.
29. William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
30. William Pugh and David Wonnacott. Eliminating false data dependences using the Omega test. In *International Conference on Programming Language Design and Implementation*, pages 140–151, June 1992.
31. Peiyi Tang. Exact side effects for interprocedural dependence analysis. In *International Conference on Supercomputing*, pages 137–146, July 1993.
32. Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures*. PhD thesis, Paris VI University, 1984.
33. Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 176–185, 1986.
34. Peng Tu and David Padua. Automatic array privatization. In *Languages and Compilers for Parallel Computing*, August 1993.

This article was processed using the LaTeX macro package with LLNCS style