IN and OUT Array Region Analyses

Béatrice CREUSILLET, creusillet@cri.ensmp.fr

Centre de Recherche en Informatique, École Nationale Supérieure des Mines de Paris,
35, rue Saint-Honoré, 77305 FONTAINEBLEAU Cedex, FRANCE.

Abstract

In order to perform powerful program optimizations, an exact interprocedural analysis of array data flow is needed. For that purpose, two new types of array region are introduced. IN and OUT regions represent the sets of array elements, the values of which are imported to or exported from the current statement or procedure. Among the various applications are: compilation of communications for message-passing machines, array privatization, compile-time optimization of local memory or cache behavior in hierarchical memory machines.

Introduction

The optimization of scientific programs for distributed memory machines, hierarchical memory machines or fault-tolerant computing environments is a particularly troublesome problem that requires a precise intra- and inter-procedural analysis of array data flow.

A recent study [8] 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 [12, 6], to the detriment of the accuracy of the results. Furthermore, this method has not yet been extended to interprocedural analyses, and its complexity makes it unpracticable on large programs.

Another approach is to calculate conservative summaries of the effects of procedure calls on sets of array elements [16, 4]. They allow the analysis of large programs, thanks to their weak complexity (in practice). 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 handle powerful optimizations.

In PIPS [10], the Interprocedural Parallelizer of Scientific Programs developed at École des Mines de Paris, we have extended Triolet's array regions [16] to calculate summaries that exactly represent the effects of statements and procedures upon sets of array elements [2], whereas the regions originally defined by Triolet were *over-approximations* of these effects. The resulting `READ/WRITE` regions are already used to efficiently compile HPF [5]. However, they cannot be used to compute array data flow, and are thus insufficient for other 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. For a massively parallel machine, these regions could be used to

calculate the communications before and after the execution of a piece of code. They can also be used to privatize array sections [2]. For a hierarchical memory machine, they provide the sets of array elements that are used or reused, and hence should be prefetched (IN regions) or kept (OUT regions) in caches or local memories [13]; the array elements that do not appear in these sets and that are accessed in the current piece of code, are only temporaries, and should be handled as such. For fault-tolerant systems, *checkpointing* [11] is a software solution that regularly saves the current state: as they provide the set of elements that will be used in further computations, IN and OUT regions could be used to reduce the amount of data to be saved.

This article is organized as follows. In Section 1, we present an example that motivates the use of IN and OUT regions, and highlights the main difficulties of their computation. Some necessary background is reviewed in Section 2. Exact regions are then defined in Section 3, along with the operators to manipulate them. Section 4 introduces IN and OUT regions. The details of their computation for the main structures of the FORTRAN language are then presented in Sections 5 and 6. We conclude in Section 7.

1 Motivating example

In the remainder of this article, we shall consider the contrived FORTRAN program of Figure 1 to illustrate the main features of the computation of IN and OUT regions.

```
1. K = FOO()                                          SUBROUTINE INC1(I)
2. DO I = 1,N                                         I = I + 1
3.    DO J = 1,N                                      END
4.       WORK(J,K) = J + K
      ENDDO
5.    CALL INC1(K)
6.    DO J = 1,N
7.       WORK(J,K) = J*J - K*K
8.       A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
    ENDDO
  ENDDO
```

Figure 1: Sample program

The purpose is to prove 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 the array 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 execution of the I loop). An optimizing compiler would conclude that the part of the array WORK that is handled in the I loop could be privatized.

There are two main difficulties in our example. First, different elements of WORK are referenced in several instructions. We shall need a way to summarize these accesses within a single representation, by using a sort of union operator. 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 instruction 6. We shall need a way to modelise this modification in order to obtain representations that depend on

the same value of K, and hence can be merged (for instance).

The next two sections present the tools we shall use to perform the analysis of our example.

2 Transformers and Preconditions

In PIPS the parallelization process is divided into several phases, either analyses or program transformations. *Intraprocedural analyses* are performed on the *hierarchical control flow graph* of the routines. The nodes of these graphs correspond to the FORTRAN language structures (DO, IF, sequence of instructions, assignment, call, ...), or are themselves small control flow graphs, when a fragment of code is unstructured (use of GOTOs). *Interprocedural analyses* propagate information over the program *call graph*. This graph is assumed acyclic, and the analyses can be performed bottom-up or top-down.

Two kinds of analyses are of interest for the remainder of this paper: *transformers* and *preconditions* [9].

Transformers abstract the effects of instructions upon the values of integer scalar variables by giving the affine relations that exist between their values before and after the execution of a statement or procedure call. In equations they are denoted by the letter 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. In Figure 2, each instruction is preceded by its corresponding transformer. A predicate with no constraint (Instruction 1) indicates that the transformation could not be represented by a set of linear constraints. When it is computable, the transformer of a loop gives its invariant (Instruction 2).

Preconditions are predicates over integer scalar variables. They hold just before the execution of the corresponding instruction. In Figure 2, 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 backward, while preconditions are propagated forward. 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 $P_2 = T_1(P_1)$.

3 Regions: definitions and operators

Array regions are sets of array elements described by equalities and inequalities defining a convex polyhedron [16]. Two other characteristics have been introduced in PIPS to represent the effects of statements and procedures upon array elements:

- the *action* upon the elements of the region; READ (R) for a use or WRITE (W) for a definition;
- the *approximation* of the region; MAY if the region is an over-approximation of the set of array elements actually referenced in the corresponding piece of code; MUST if the region exactly represents this set (*exact* region).

```
   C  P() {}
   C  T(K) {}
1.       K = FOO()
   C  P(K) {}
   C  T(I,K) {K+1==K#init+I}
2.       DO I = 1,N
   C  P(I,K) {1<=I, I<=N}
3.          DO J = 1,N
   C  P(I,J,K) {1<=I, I<=N, 1<=J, J<=N}
4.             WORK(J,K) = J + K
            ENDDO
   C  P(I,K) {1<=I, I<=N}
   C  T(K) {K==K#init+1}
5.          CALL INC1(K)
6.          DO J = 1,N
   C  P(I,J,K) {1<=I, I<=N, 1<=J, J<=N}
7.             WORK(J,K) = J*J - K*K
8.             A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
            ENDDO
         ENDDO
```

Figure 2: Transformers and preconditions

For instance, the region:

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

where ϕ_1 et ϕ_2 respectively represent the first and second dimensions of `A`, corresponds to a definition of the element `A(I,I)`.

In order to manipulate regions and propagate them along the control flow graph of the program, we need several binary operators: union ($\overline{\cup}$), intersection ($\cap$), and difference ($\ominus$). We also need unary operators ($\mathcal{T}_{\sigma_1 \to \sigma_2}$, $\mathcal{T}_{\sigma_1 \to \sigma_2 \setminus I}$ and $proj_I$) to deal with variables of the program that are modified. These operators are presented below.

Union

The union operator is used to merge two regions. Unfortunately, the union of two convex polyhedra is not necessarily a convex polyhedron. The operator $\overline{\cup}$ we use instead is the convex hull. It may contain points that do not belong to the original polyhedra. Thus, the resulting region is a `MAY` region. The third column in Table 1 gives the approximation of the resulting region against the characteristics of the initial regions.

Intersection

The intersection of two convex polyhedra is a convex polyhedron. It follows that the intersection of two `MUST` regions is a `MUST` region. The impact of the approximations of the initial regions is summarized in the fourth column of Table 1.

R_1	R_2	$R_1 \overline{\cup} R_2$	$R_1 \cap R_2$	$R_1 \ominus R_2$	
MUST	MUST	MUST, iff exact convex hull	MUST	$R_1 - R_2$,	MUST iff exact
MUST	MAY	MUST, iff $R_2 \subseteq R_1$	MAY	R_1,	MUST iff $R_1 \cap R_2 = \varnothing$
MAY	MUST	MUST, iff $R_1 \subseteq R_2$	MAY	$R_1 - R_2$,	MAY
MAY	MAY	MAY	MAY	R_1,	MAY

Table 1: Binary operators on regions

Difference

The difference of two convex polyhedra is not necessarily a convex polyhedron. The chosen operator (denoted $\ominus$) gives an over-approximation of the actual difference of the original regions. Its features are described in Table 1, Column 5.

Translation from one store to another

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 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. We shall call $\mathcal{T}_{\sigma_1 \to \sigma_2}$ the transformation of a region relative to the store σ_1 into a region relative to the store σ_2.

This transformation has been described in [2]. 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 the 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#init} the predicate of a region relative to σ_1. We first add the transformer corresponding to the statement (T(I) {I==I#init+1}). This gives the predicate {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 elimination of a variable from a convex polyhedron may introduce integer points that do not belong to the actual projection. Ancourt or Pugh [1, 14] have proposed sufficient conditions under which this elimination is exact. They do not apply in our case because program entities must be considered as parameters (symbolic constants) for the polyhedron, and not as variables.

For instance, the elimination of I from {ϕ_1==I, I<=J} gives {ϕ_1<=J}. It satisfies the usual conditions for an exact projection. However, these two polyhedra do not describe the same region. The first contains only one element, uniquely defined by the value of I in the current store, even if we do not know it. Whereas the second contains all the elements such that ϕ_1 is smaller than the current value of J.

On the contrary, the elimination of I from $\{2\phi_1$==3I, 2I==J$\}$, which gives $\{4\phi_1$==3J$\}$, is usually considered inexact. In our case, and referring to a given store, ϕ_1 is uniquely defined in both cases by the values of the variables appearing in both polyhedra.

Since this is not the purpose of our paper, we refer the reader to [2] for sufficient conditions of the exact projection of a region polyhedron along a program variable, and their verification. When these new conditions are not verified, the elimination is performed, but the original region becomes a MAY region.

Sometimes, we shall need to eliminate all but one variable (e.g. I). We shall denote the corresponding transformation $\mathcal{T}_{\sigma_1 \rightarrow \sigma_2 \setminus I}$.

Elimination of loop indices

During the propagation of regions in a linear block of instructions, a loop index is considered as a normal program entity. It receives a particular treatment only when propagating regions outward or inward the corresponding loop, because it then takes successively a whole range of values (iteration space), instead of a single one. Thus, the elimination of a loop index from the polyhedron of a region is exact if the following conditions are verified:

1. the expressions of the lower and upper bounds, and of the increment, are affine;
2. the absolute value of the increment is equal to 1;
3. the conditions of Ancourt or Pugh for an exact projection are verified.

The first two conditions ensure that the iteration space can be described exactly by a convex polyhedron over the program entities (i.e. $lb \leq i \leq ub$ if lb and ub are respectively the upper and lower bounds of the loop index). We shall denote $proj_I$ the elimination of loop index I.

Constraining the regions predicate

In order to have more information on ϕ variables, the constraints of the preconditions are systematically added to the predicate of the region. This is particularly useful when merging two regions. For instance, $\{\phi_1$==I$\} \cup \{\phi_1$==J$\} = \{\}$. If we add the current preconditions (e.g. {I==J}) to the original regions, we obtain $\{\phi_1$==I,I==J$\}$ instead of {}. This operation increases the accuracy of the analysis, without modifying the definition of regions.

The purpose of this paper is not to describe the computation of READ and WRITE regions. However, since we shall need them for the calculation of IN and OUT regions, we provide those concerning the array WORK in our current example, in Figure 3. Each elementary instruction is preceded by its READ and WRITE regions; and the regions preceding a DO are the summary regions for the loop. Notice that the variable K is modified in the body of the I loop. Therefore, K does not represent the same value in the regions preceding and following the call to INC1.

```
1.       K = FOO()
   C <WORK(φ1,φ2)-W-MUST-{1<=φ1, φ1<=N, K<=φ2, φ2<=K+N}>
   C <WORK(φ1,φ2)-R-MUST-{1<=φ1, φ1<=N, K<=φ2, φ2<=K+N}>
2.       DO I = 1,N
   C <WORK(φ1,φ2)-W-MUST-{1<=φ1, φ1<=N, φ2==K, 1<=I, I<=N}>
3.          DO J = 1,N
   C <WORK(φ1,φ2)-W-MUST-{φ1==J, φ2==K, 1<=J, J<=N, 1<=I, I<=N}>
4.             WORK(J,K) = J + K
            ENDDO
5.          CALL INC1(K)
   C <WORK(φ1,φ2)-W-MUST-{1<=φ1, φ1<=N, φ2==K, 1<=I, I<=N}>
   C <WORK(φ1,φ2)-R-MUST-{1<=φ1, φ1<=N, K-1<=φ2, φ2<=K, 1<=I, I<=N}>
6.          DO J = 1,N
   C <WORK(φ1,φ2)-W-MUST-{φ1==J, φ2==K, 1<=J, J<=N, 1<=I, I<=N}>
7.             WORK(J,K) = J*J - K*K
   C <WORK(φ1,φ2)-R-MUST-{φ1==J, K-1<=φ2, φ2<=K, 1<=J, J<=N, 1<=I, I<=N}>
8.             A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
            ENDDO
         ENDDO
```

Figure 3: READ/WRITE regions

4 IN and OUT Regions

READ and WRITE regions summarize the exact effects of statements and procedures upon array elements. However, they do not represent the flow of array elements, the knowledge of which is necessary to achieve many optimizations. For that purpose, we introduce two new types of region: IN and OUT regions.

IN regions contain the array elements, the values of which are (MUST) or may be (MAY) *imported* by the current piece of code. These are the elements that are read before being possibly redefined by another instruction of the same fragment. In Figure 1, in the body of the second J loop, the element WORK(J,K) is read, but its value is not imported 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 (MUST) or may be (MAY) used afterwards, in the execution order of the program. 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 loop I.

The *interprocedural* propagation of IN regions is similar to that of READ/WRITE regions [9, 7]. It is a backward propagation: IN regions corresponding to formal parameters are translated into regions corresponding to actual parameters, at each call site. On the contrary, the propagation of OUT regions on the call graph is a forward propagation: actual parameters are translated into formal ones at each call site, and the resulting regions are merged into a unique summary, which forms the OUT regions of the procedure. For both types of region, the propagation consists in adding to the predicate of the region the equation giving the relations between the *subscript values* of the arrays in terms of ϕ variables. The ϕ variables of the formal (resp. actual) array

in case of IN (resp. OUT) regions are then eliminated. This can be optimized by considering the similar dimensions of both arrays. This method handles array reshaping, which was only partially treated in [15].

We now limit ourselves to the intraprocedural computation of IN and OUT regions. For that purpose, we use the following notations. If S_k is the k-th instruction of a sequence of complex instructions, σ_k and σ_{k+1} are the memory stores immediately preceding and following it; and W_k, IN_k, and OUT_k are respectively its WRITE, IN and OUT regions relative to the store σ_k.

5 Computation of IN regions

In this section, we describe the calculation of IN regions of an assignment, a sequence of complex instructions, or a loop. We assume that the code is correct, i.e. that no variable or part of a variable is used without being initialized. This assumption is often made in parallelizing compilers, where the purpose is not to verify programs, but to transform and optimize them.

5.1 Assignment

For an assignment, IN regions are identical to READ regions because the values of the referenced elements cannot come from the assignment itself. In the statement 8 (A(I) = A(I)+WORK(J,K-1)+WORK(J,K)) in the program of Figure 1, the elements that are read are WORK(J,K-1), WORK(J,K) and A(I). These are also the elements, the values of which are imported. Thus, the IN regions are:

<WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, K-1<=ϕ_2, ϕ_2<=K, 1<=I, I<=N , 1<=J, J<=N}>
<A(ϕ_1)-IN-MUST-{ϕ_1==I, 1<=I, I<=N , 1<=J, J<=N}>

5.2 Sequence of instructions

Let us first take an example. We consider the body of the second J loop in Figure 1, which consists in the sequence of instructions 7 and 8. The instruction 8 imports two elements of the array WORK: WORK(J,K) and WORK(J,K-1). This is represented by the region[1]:

<WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, K-1<=ϕ_2, ϕ_2<=K}>

One of these elements, WORK(J,K), is initialized in the instruction 7. Since the instruction 7 imports no array element, the sequence of instructions does not import the value of WORK(J,K), and the IN region of the sequence is finally obtained by removing the elements of the WRITE region of the instruction 7 from the IN region of the instruction 8:

<WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, K-1<=ϕ_2, ϕ_2<=K}>
$\ominus$ <WORK(ϕ_1,ϕ_2)-W-MUST-{ϕ_1==J, ϕ_2==K}>
= <WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, ϕ_2==K-1}>

The IN region of the sequence of instructions 7 and 8 only contains the element WORK(J,K-1).

[1]Without the constraints from the preconditions, since integer scalar variables are not modified in this loop body.

We now give the general method. We are interested in the region IN_B corresponding to the block (or sequence) of instructions $B = S_1, \ldots, S_n$, and relative to the store $\sigma_B = \sigma_1$ preceding the execution of B. It is the set of array elements that are read by the instructions of B, and the values of which come from the instructions preceding B.

For each instruction S_k, the corresponding regions W_k and IN_k, relative to the store σ_k preceding S_k are supposed to be known.

We denote IN'_k ($k \in [1, n]$) the IN regions corresponding to the subsequence $S_k, \ldots S_n$. IN_B is then defined by:

$$\begin{cases} IN'_n = IN_n \\ IN'_k = IN_k \,\overline{\cup}\, [\, \mathcal{T}_{\sigma_{k+1} \to \sigma_k}(IN'_{k+1}) \,\ominus\, W_k \,] \\ IN_B = IN'_1 \end{cases}$$

$\mathcal{T}_{\sigma_{k+1} \to \sigma_k}$ translates the IN regions corresponding to $S_{k+1}, \ldots, S_n$ in the same store σ_k as the WRITE and IN regions of S_k. $\mathcal{T}_{\sigma_{k+1} \to \sigma_k}(IN'_{k+1}) - W_k$ represents the regions imported by the subsequence $S_{k+1}, \ldots, S_n$, but not defined by the statement S_k. Merged with IN_k, they give the set of elements that are read by the sequence $S_k, \ldots, S_n$ before being possibly redefined by any other statement of the same subsequence.

5.3 DO loop

Our purpose is now to compute the summary IN regions of the second J loop in Figure 1, given the WRITE and IN regions of its body at iteration J:

<WORK(ϕ_1,ϕ_2)-W-MUST-{ϕ_1==J, ϕ_2==K, 1<=I, I<=N, 1<=J, J<=N}>
<WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, ϕ_2==K-1, 1<=I, I<=N, 1<=J, J<=N}>

The previous IN region represents the array elements that are imported by the iteration J from the instructions preceding the loop, and from the preceding iterations (J′ such that {J′<=J-1}). We must first remove those elements that are imported from the previous iterations, hence those that are written by these iterations. The next region represents the elements written by a single iteration J′ preceding J:

<WORK(ϕ_1,ϕ_2)-W-MUST-{ϕ_1==J′, ϕ_2==K, 1<=J′, J′<=J-1, J<=N, 1<=I, I<=N}>

By eliminating the loop index J′, we obtain the set of array elements written by *all* the iterations preceding the iteration J:

<WORK(ϕ_1,ϕ_2)-W-MUST-{1<=ϕ_1, ϕ_1<=J-1, ϕ_2==K, J<=N, 1<=I, I<=N}>

We now remove these elements from the set of elements imported by the iteration J:

<WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, ϕ_2==K-1, 1<=I, I<=N, 1<=J, J<=N}>
$\ominus$ <WORK(ϕ_1,ϕ_2)-W-MUST-{1<=ϕ_1, ϕ_1<=J-1, ϕ_2==K, J<=N, 1<=I, I<=N}>
$=$ <WORK(ϕ_1,ϕ_2)-IN-MUST-{ϕ_1==J, ϕ_2==K-1, 1<=I, I<=N, 1<=J, J<=N}>

This last region represents the set of elements imported by the iteration J from the instructions preceding the loop. We then eliminate J to obtain the set of elements imported by *all* the iterations from the instructions preceding the loop:

<WORK(ϕ_1,ϕ_2)-IN-MUST-{1<=ϕ_1, ϕ_1<=N, ϕ_2==K-1, 1<=I, I<=N}>

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

We now give the general equations to compute the regions IN_L corresponding to the DO loop L, given the WRITE and IN regions of its body. They are not only functions of the value i of the loop index I, but also of the variables modified in the loop body (collectively denoted v). Hence, we denote them $W(i,v)$ and $IN(i,v)$ for iteration i.

v variables are themselves functions of the loop index. We must eliminate them to obtain WRITE and IN 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 the operator $\mathcal{T}_{\sigma_L \rightarrow \sigma_i \setminus I}$, σ_L being the store preceding the loop L, and σ_i the store preceding the iteration such that $I = i$. This operator is based on the transformer of the loop body, which represents the loop invariant (when it is computable). In order to simplify the next equation, we denote:

$$\begin{aligned} W(i) &= \mathcal{T}_{\sigma_L \rightarrow \sigma_i \setminus I}(\ W(i,v)\ ) \\ IN(i) &= \mathcal{T}_{\sigma_L \rightarrow \sigma_i \setminus I}(\ IN(i,v)\ ) \end{aligned}$$

The IN regions of a DO loop are then given by the following equation:

$$IN_L = proj_I[\ IN(i) \ominus proj_{I'}(W(i')_{s.i' < s.i})\ ]$$

where s is the sign of the loop increment.

$proj_{I'}(W(i')_{s.i' < s.i})$ represents the array elements written by all the iterations i' preceding the iteration i. Thus, $IN(i) \ominus proj_{I'}(W(i')_{s.i' < s.i})$ contains the elements imported by the iteration i, but not from the preceding iterations. And finally, $proj_I[\ IN(i) \ominus proj_{I'}(W(i')_{s.i' < s.i})\ ]$ is the set of array elements imported by all the iterations of the loop from the instructions preceding it.

6 Computation of OUT regions

The computation of OUT regions is a forward analysis. Thus, we do not calculate the OUT regions of an instruction, a sequence of instructions, or a loop, but the OUT regions of the instructions of a sequence (given the OUT regions of the sequence), or of the body of a loop (given the OUT regions of the loop). The OUT region corresponding to the whole program is the empty set[2].

6.1 Instructions of a sequence

As an illustration, we consider the body of the second J loop in Figure 1. We assume that its OUT region concerning the array WORK is the empty set. We want to calculate the set of elements of the array WORK that are exported by the instructions 7 and 8.

The array elements exported by the instruction 8 are those that it defines (here $\varnothing$ for the array WORK), and that are exported by the whole sequence ($\varnothing$): this instruction exports no element of the array WORK.

The instruction 7 may export the elements it defines toward the instructions following the execution of the whole sequence: since the latter exports no elements, neither

[2] I/Os are performed by the program itself, and they are taken into account during the computation of OUT regions.

does the instruction 7. It may also export the elements it defines toward the instruction 8: these are the element written by the instruction 7 and imported by the instruction 8 (not just the written elements because the instruction 8 might be a complex instruction):

$$\begin{array}{ll} & \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-MUST-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>} \\ \cap & \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-MUST-\{}\phi_1\texttt{==J, K-1<=}\phi_2\texttt{, }\phi_2\texttt{<=K\}>} \\ = & \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-MUST-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>} \end{array}$$

The first instruction exports the element it defines (`WORK(J,K)`) towards the second one.

We now consider the general problem. The region OUT_B corresponding to the sequence $B = S_1, \ldots, S_n$, and relative to the store $\sigma_B = \sigma_1$ preceding it, is supposed to be known. We are interested in the regions OUT_k corresponding to each instruction S_k. These regions contain the array elements written by the current instruction, and the values of which are used by the next statements, inside and outside the sequence.

We call OUT'_k the set of array elements defined by the subsequence $S_1, \ldots, S_k$, and the values of which are used *after* the execution of B (i.e. are exported outside of B). The equations defining OUT_k are then:

$$\left\{ \begin{array}{l} OUT'_n = \mathcal{T}_{\sigma_B \rightarrow \sigma_n}(OUT_B) \\ OUT_n = W_n \cap OUT'_n \\ OUT'_k = \mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(OUT'_{k+1} \ \ominus \ W_{k+1}), \ \forall\, k \in [1..n-1] \\ OUT_k = W_k \cap [\, OUT'_k \,\overline{\cup}\, \mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(IN'_{k+1})\,], \ \forall\, k \in [1..n-1] \end{array} \right.$$

OUT_B is the `OUT` region corresponding to $B = S_1, \ldots, S_n$. It is relative to the store σ_B preceding its execution. Therefore, $\mathcal{T}_{\sigma_B \rightarrow \sigma_n}(OUT_B)$ is the region corresponding to $S_1, \ldots, S_n$, but relative to the store σ_n preceding S_n. This is exactly the definition of OUT'_n. Thus $OUT'_n = \mathcal{T}_{\sigma_B \rightarrow \sigma_n}(OUT_B)$.

The elements exported by the subsequence $S_1, \ldots, S_k$ towards the outside of B, are those that exported by the subsequence $S_1, \ldots, S_{k+1}$ towards the outside of B, minus the elements exported by the instruction S_{k+1}. The corresponding `OUT` regions must be relative to the store σ_k, which gives:

$$OUT'_k = \mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(OUT'_{k+1} - W_{k+1})$$

Finally, the `OUT` regions of the instruction S_k are the regions written by S_k and exported towards the outside of B ($W_k \cap OUT'_k$), to which we must add the regions written by S_k and exported towards $S_{k+1}, \ldots, S_n$ ($W_k \cap \mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(IN'_{k+1})$). This gives:

$$OUT_k = W_k \cap [\, OUT'_k \,\overline{\cup}\, \mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(IN'_{k+1})\,]$$

6.2 Body of a DO loop

First, let us take a simple example. We consider the `I` loop in Figure 1. We want to compute the `OUT` regions of its body concerning the array `WORK`, assuming that the `OUT` region of the loop is the empty set and given the `WRITE` and `IN` regions of the body[3]:

[3] `{0==1}` represents the empty set.

<WORK(ϕ_1,ϕ_2)-W-MUST-{1<=ϕ_1, ϕ_1<=N, K<=ϕ_2, ϕ_2<=K+1, 1<=I, I<=N}>
<WORK(ϕ_1,ϕ_2)-IN-MUST-{0==1}>

We first compute the set of array elements exported by the iteration I towards the outside of the loop: it is the empty set, because the OUT region of the loop is the empty set.

We then compute the set of array elements exported by the iteration I toward the subsequent iterations I′ ({I+1<=I′}). These are the elements written by the iteration I, and:

1. imported by the subsequent iterations I′;
2. but not defined by the iterations I″ between the iteration I and the iteration I′.

This last constraint ensures that, among the elements read by the iteration I′, we do not keep the elements that are imported from the iterations between I and I′; but we keep those that may come from the iteration I.

Here, the IN regions of the loop body are the empty set. Thus, the iteration I does not exports the elements it defines toward the subsequent iterations: its OUT region is the empty set.

We now consider the general case. We assume that we know the OUT regions OUT_L corresponding to the loop L. We want to calculate the OUT regions of its body, $OUT(i)$, if i is the value of the loop index I, given its WRITE and IN regions, $W(i)$ and $IN(i)$ (we have seen in section 5.3 how to get rid of variables that vary in the loop body).

The loop body represents any iteration i. Its OUT regions contain the elements that are defined during this iteration, and that are used, either in the next iterations, or outside the loop. This gives the following equation:

$$\begin{aligned} OUT(i) \;=\; & \{\, [\, W(i) \ominus proj_{I'}(W(i')_{s.i'>s.i}) \,] \cap \mathcal{T}_{\sigma_L \to \sigma_i}(OUT_L) \,\} \;\overline{\cup} \\ & \{\, W(i) \cap [\, proj_{I'}(IN(i')_{s.i'>s.i} \ominus proj_{I''}(W(i'')_{s.i<s.i''<s.i'})) \,] \,\} \end{aligned}$$

where s represents the sign of the loop increment.

$proj_{I'}(W(i')_{s.i'>s.i})$ contains the array elements that are defined by all the iterations i' posterior to i. Thus, $W(i) \ominus proj_{I'}(W(i')_{s.i'>s.i})$ contains the elements that are defined by iteration i, and that are not defined in the subsequent iterations. $\mathcal{T}_{\sigma_L \to \sigma_i}(OUT_L)$ represents the OUT regions of loop L, but relative to the store preceding the iteration i. Then, $[\, W(i) \ominus proj_{I'}(W(i')_{s.i'>s.i}) \,] \cap \mathcal{T}_{\sigma_L \to \sigma_i}(OUT_L)$ contains the array elements that are defined by the iteration i *and* that are exported towards the outside of the loop.

Similarly, $proj_{I''}(W(i'')_{s.i<s.i''<s.i'})$ represents the elements that are written by all the iterations between iterations i and i'. Then, $IN(i')_{s.i'>s.i} \ominus proj_{I''}(W(i'')_{s.i<s.i''<s.i'})$ contain the elements imported by iteration i' but that do not come from any iteration posterior to i, i.e. that are imported either from the instructions preceding the loop or from iterations up to iteration i (included). The projection along i' gives the set of elements that are imported by all iterations following iteration i, either from iteration i and its predecessors, or from the outside of the loop. Thus, $W(i) \cap [\, proj_{I'}(IN(i')_{s.i'>s.i} \ominus proj_{I''}(W(i'')_{s.i<s.i''<s.i'})) \,]$ exactly represents the elements that are defined by the iteration i and that are effectively used in the subsequent iterations.

Note: In order to ensure the precision of the analysis, it is necessary to add to the predicate of $\mathcal{T}_{\sigma_L \to \sigma_i}(OUT_L)$ the preconditions induced by the variation domain of the loop index, i.e. $lb \leq i \leq ub$ if lb and ub are respectively its lower and upper bounds.

We now take another example to illustrate some features of the previous algorithm. We consider the `I` loop in the program of figure 1. We assume that the `OUT` regions of the loop concerning the array `A` are:

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

We first calculate $\mathcal{T}_{\sigma_L \to \sigma_i}(OUT_L)$. We add the constraints of the loop transformer, `T(I,K){K==K#INIT+I-1}`, to the polyhedron of the region, along with the conditions on the variation domain of `I`, i.e. `{1<=I, I<=N}`. This leads to:

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

Given that the `WRITE` and `IN` regions of the loop body concerning the array `A` are:

$$\texttt{<A(}\phi_1\texttt{)-W-MUST-\{}\phi_1\texttt{==I, 1<=I, I<=N\}>}$$
$$\texttt{<A(}\phi_1\texttt{)-IN-MUST-\{}\phi_1\texttt{==I, 1<=I, I<=N\}>}$$

we successively have:

$$\begin{array}{lcl}
W(i')_{s.i'>s.i} & = & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{}\phi_1\texttt{==I', 1<=I, I+1<=I', I'<=N\}>} \\
proj_{I'}(W(i')_{s.i'>s.i}) & = & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{I+1<=}\phi_1\texttt{, }\phi_1\texttt{<=N, 1<=I\}>} \\
W(i) \ominus proj_{I'}(W(i')_{s.i'>s.i}) & = & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{}\phi_1\texttt{==I, 1<=I, I<=N\}>} \\
[\, W(i) \ominus proj_{I'}(W(i')_{s.i'>s.i}) \,] & \cap & \mathcal{T}_{\sigma_L \to \sigma_i}(OUT_L) \; = \\
 & & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{}\phi_1\texttt{==I, 1<=I, I<=N\}>} \\
 & \cap & \texttt{<A(}\phi_1\texttt{)-OUT-MUST-\{1<=}\phi_1\texttt{, }\phi_1\texttt{<=N, 1<=I, I<=N\}>} \\
 & = & \texttt{<A(}\phi_1\texttt{)-OUT-MUST-\{}\phi_1\texttt{==I, 1<=I, I<=N\}>}
\end{array}$$

This last region represents the elements exported outward the loop. We now calculate the set of elements exported towards the other iterations:

$$\begin{array}{lcl}
W(i'')_{s.i<s.i''<s.i'} & = & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{}\phi_1\texttt{==I'',1<=I,I+1<=I'',I''+1<=I',I'<=N\}>} \\
proj_{I''}(W(i'')_{s.i<s.i''<s.i'}) & = & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{I+1<=}\phi_1\texttt{, }\phi_1\texttt{+1<=I', I'<=N, 1<=I\}>} \\
IN(i')_{s.i'>s.i} & = & \texttt{<A(}\phi_1\texttt{)-IN-MUST-\{}\phi_1\texttt{==I', 1<=I, I+1<=I', I'<=N\}>} \\
IN(i')_{s.i'>s.i} \ominus proj_{I''}(W(i'')_{s.i<s.i''<s.i'}) = & & \\
 & & \texttt{<A(}\phi_1\texttt{)-IN-MUST-\{}\phi_1\texttt{==I', 1<=I, I+1<=I', I'<=N\}>} \\
proj_{I'}(IN(i')_{s.i'>s.i} \ominus proj_{I''}(W(i'')_{s.i<s.i''<s.i'})) = & & \\
 & & \texttt{<A(}\phi_1\texttt{)-IN-MUST-\{I+1<=}\phi_1\texttt{, }\phi_1\texttt{<=N, 1<=I\}>} \\
W(i) \cap [\, proj_{I'}(IN(i')_{s.i'>s.i} \ominus proj_{I''}(W(i'')_{s.i<s.i''<s.i'})) \,] = & & \\
 & & \texttt{<A(}\phi_1\texttt{)-W-MUST-\{}\phi_1\texttt{==I, 1<=I, I<=N\}>} \\
 & \cap & \texttt{<A(}\phi_1\texttt{)-IN-MUST-\{I+1<=}\phi_1\texttt{, }\phi_1\texttt{<=N, 1<=I\}>} \\
 & = & \varnothing
\end{array}$$

Thus, the iteration i exports no element of `A` towards the subsequent iterations. And finally,

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

We provide in Figure 4 the `IN` and `OUT` regions of the program of Figure 1, given the `OUT` regions of the outer loop.

```
1.       K = FOO()
   C <A(φ1)-IN-MUST-{1<=φ1, φ1<=N}
   C <A(φ1)-OUT-MUST-{1<=φ1, φ1<=N}
2.       DO I = 1,N
   C <WORK(φ1,φ2)-OUT-MUST-{1<=φ1, φ1<=N, φ2==K, 1<=I, I<=N}>
3.          DO J = 1,N
   C <WORK(φ1,φ2)-OUT-MUST-{φ1==J, φ2==K, J<=I, J<=N, 1<=I, I<=N}>
4.             WORK(J,K) = J + K
            ENDDO
5.          CALL INC1(K)
   C <A(φ1)-OUT-MUST-{φ1==I, 1<=I, I<=N}
   C <WORK(φ1,φ2)-IN-MUST-{1<=φ1, φ1<=N, φ2==K-1, 1<=I, I<=N}>
6.          DO J = 1,N
   C <WORK(φ1,φ2)-OUT-MUST-{φ1==J, φ2==K, J<=I, J<=N, 1<=I, I<=N}>
7.             WORK(J,K) = J*J - K*K
   C <A(φ1)-IN-MUST-{φ1==I, 1<=I, I<=N, J<=I, J<=N}
   C <A(φ1)-OUT-MUST-{φ1==I, 1<=I, I<=N, J<=I, J<=N}
   C <WORK(φ1,φ2)-IN-MUST-{φ1<==J, K-1<=φ2, φ2<=K, J<=I, J<=N, 1<=I, I<=N}>
8.             A(I) = A(I)+WORK(J,K)+WORK(J,K-1)
            ENDDO
         ENDDO
```

Figure 4: IN and OUT regions

7 Conclusion

We have introduced two new types of array regions that can be used to compute the flow of array elements and perform powerful optimizations. IN and OUT regions represent the sets of array elements that are imported or exported by the corresponding piece of code. Their intraprocedural propagation relies on a preliminary analysis of the effects of statements upon integer scalar variables. This locally ensures the accuracy of the analyses, even when variables occurring in regions expressions are modified. The current implementation covers all the intraprocedural structures of the FORTRAN language, along with the interprocedural propagation [2, 7].

Experiments performed on some of the Perfect Club benchmarks [3] have shown the practicability of these analyses in spite of the theoretically exponential complexity of the operators on polyhedra. However, further experiments are needed to determine if the representation of regions in polyhedral form is precise enough to allow optimizations such as array privatization, generation of communications in distributed memory machines, compile-time optimization of cache behavior in hierarchical memory machines, or improvement of the use of distributed shared memory systems.

A more powerful alternative to polyhedra would be the use of finite unions of polyhedra. The cost, both in terms of memory use and computation time, would certainly be more important. But it would increase the accuracy of analyses by avoiding inexact unions or differences.

Acknowledgments

I am thankful to Corinne Ancourt, Fabien Coelho, François Irigoin, Pierre Jouvelot and Alexis Platonoff for their careful reading, and their helpful comments.

References

[1] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Symposium on Principles and Practice of Parallel Programming*, April 1991.

[2] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux (exact array region analysis and array privatization). Master's thesis, Université Paris VI, France, September 1994. Available via `http://www.cri.ensmp.fr/~creusil`.

[3] M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, Center for Supercomputing Research and Development, University of Illinois, May 1989.

[4] 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.

[5] Fabien Coelho. Compilation of I/O communications for HPF. In *Frontiers'95*, February 1995. Available via `http://www.cri.ensmp.fr/~coelho`.

[6] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.

[7] Béatrice Creusillet and François Irigoin. Interprocedural array regions analyses. Technical report A-270, CRI, École des Mines de Paris, March 1995. Submitted to LCPC'95.

[8] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, September 1991.

[9] François Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, September 1992.

[10] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : An overview of the PIPS project. In *International Conference on Supercomputing*, June 1991.

[11] 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.

[12] Vadim Maslov. Lazy array data-flow analysis. In *Symposium on Principles of Programming Language*, pages 311–325, January 1994.

[13] Ravi Mirchandaney, Seema Hiranandani, and Ajay Sethi. Improving the performance of DSM systems via compiler involvement. In *International Conference on Supercomputing*, pages 763–772, November 1994.

[14] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[15] Rémi Triolet. Interprocedural analysis for program restructuring with parafrase. Technical report 538, Center for Supercomputing Research and Development, University of Illinois, December 1985.

[16] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *Proceedings of the ACM Symposium on Compiler Construction*, 1986.