Interprocedural Array Region Analyses

Béatrice CREUSILLET, François IRIGOIN*

Ecole des Mines de Paris/CRI

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, that handles array reshapes.

Introduction

The efficient compilation of scientific programs for massively parallel machines, hierarchical memory machines or fault-tolerant computing environments requires a precise intra- and inter-procedural analysis of array data flow.

A recent type of analysis [9] 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 [14, 8], to the detriment of the accuracy of the results. Furthermore, this method does not allow interprocedural analyses, and its complexity makes it unpracticable for large programs.

Another approach is to calculate conservative summaries of the effects of procedure calls on sets of array elements [19, 6]. They allow the analysis of large programs, thanks to their low 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 [12], the Interprocedural Parallelizer of Scientific Programs developed at Ecole des Mines de Paris, we have extended Triolet's array regions [19] to calculate *exact* summaries of the effects of statements and procedures on sets of array elements [3]. The resulting `READ/WRITE` regions are already used to efficiently compile HPF [7]. 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

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

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 [3]. 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, and that are accessed in the current piece of code, are only temporaries, and should be handled as such. For fault-tolerant systems, *checkpointing* [13] 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 paper is organized as follows. In Section 1, we review the necessary background. Section 2 presents array regions and the operators to manipulate them. Their intra- and interprocedural propagation is detailled in Sections 3 and 4. `IN` and `OUT` regions are then introduced in Section 5.

1 Prerequisites and notations

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* [12] of the routines. The nodes of these graphs correspond to the FORTRAN language control structures (`DO`, `IF`, sequence of instructions, assignement, call, ...), or are themselves flat 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, according to the FORTRAN standard, and the analyses can be performed bottom-up or top-down.

Two auxiliary analyses are of interest in the remainder of this paper: *transformers* and *preconditions* [11].

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 1 shows the transformers corresponding to each instruction.

Preconditions are predicates over integer scalar variables. They hold just before the execution of the corresponding instruction. In Figure 1, 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 $P_2 = T_1(P_1)$.

```
C  T(K) {K == 3}
  K = 3
C  T(K) {}
  K = FOO(N)
C  T(K) {K == K#init + 1}
  CALL INC1(K)
```

```
C  P() {}
  K = 3
C  P(K) {K==3}
  K = FOO()
C  P(K) {}
  CALL INC1(K)
C  P(K) {}
```

Fig. 1. Transformers and preconditions

In the remainder of this paper, we shall use the following notations: for a statement S_k, σ_k is a store preceding it, and T_k its corresponding transformer.

2 Regions: definitions and operators

An array region is a set of array elements described by equalities and inequalities defining a convex polyhedron [19]. 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 overapproximation of the set of array elements actually referenced in the corresponding piece of code; `MUST` if the region exactly represents this set (*exact* region).

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

In order to summarize array accesses, and to propagate the summaries along control flow paths, we need several operators: union, intersection, difference, and unary operators.

Union The union operator is used to merge two elementary regions. Unfortunately, the union of two convex polyhedra is not necessarily a convex polyhedron. The operator $\bar{\cup}$ we use instead is the convex hull. It may generate points that do not belong to the original polyhedra. In this case, the resulting region is `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	$R_1 \bar{\cup} R_2$	$R_1 \cap R_2$	$R_1 \ominus R_2$	
MUST	MUST	MUST, iff exact convex hull	MUST	$\bar{\cup}\ (R_1 \cap \bar{R}_2)$,	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	$\bar{\cup}\ (R_1 \cap \bar{R}_2)$,	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 $\ominus$ 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 $\mathcal{T}_{\sigma_1 \rightarrow \sigma_2}$ 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 [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, $\mathcal{T}_{\sigma_k \rightarrow \sigma_{k+1}}$ and $\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}$, 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 statement `I = I + 1`, σ_2 the store following it, and $\{\phi_1$`==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 $\{\phi_1$`==I#init, I==I#init+1`$\}$. We then eliminate `I#init`, because it refers to σ_1. We obtain $\{\phi_1$`==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 [2, 16] have proposed 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 $\{\phi_1$`==I, I<=J`$\}$ gives $\{\phi_1$`<=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 $\{2\phi_1\texttt{==3I, 2I==J}\}$, which gives $\{4\phi_1\texttt{==3J}\}$, 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 [3] for conditions of the exact elimination of a symbolic constant from a polyhedron, and their verification. When these conditions are not met, 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 as $\mathcal{T}_{\sigma_1 \to \sigma_2 \backslash i}$.

Elimination of loop indices or ϕ variables 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 towards the outside or the inside of the corresponding loop, because it then successively takes 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 met:

1. the expressions of the lower and upper bounds 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 met.

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 respectively are the upper and lower bounds of the loop index). We shall designate by $proj_i$ the elimination of loop index i.

We also need to eliminate ϕ variables from regions during their interprocedural propagation. These entities are variables in the usual sense: their elimination is exact if the conditions of Ancourt or Pugh are met.

Constraining 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, $\{\phi_1\texttt{==I}\} \;\bar{\cup}\; \{\phi_1\texttt{==J}\} = \{\}$. If we add the current preconditions (e.g. $\{\texttt{I==J}\}$) to the original regions, we obtain $\{\phi_1\texttt{==I,I==J}\}$ instead of $\{\}$. This operation increases the accuracy of the analysis, without modifying the definition of regions.

3 Intraprocedural propagation

In this section, we detail the computation of `READ` and `WRITE` regions for the main structures of the FORTRAN language: assignment, sequence of complex instructions, `DO` loop. As an illustration, we shall consider the contrived example of Figure 2.

```
K = FOO()
DO I = 1,N
   DO J = 1,N
      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
```

```
SUBROUTINE INC1(I)
I = I + 1
END
```

Fig. 2. Sample program

3.1 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 (`MUST` regions) if and only if the indices expressions are affine functions of the program variables.

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

For instance, for the instruction `A(I) = A(I)+WORK(J,K)+WORK(J,K-1)` in our working example, elementary `READ` regions are:

```
<A(φ1)-R-MUST-{φ1==I, 1<=I<=N, 1<=J<=N}>
<WORK(φ1,φ2)-R-MUST-{φ1==J, φ2==K, 1<=I<=N, 1<=J<=N}>
<WORK(φ1,φ2)-R-MUST-{φ1==J, φ2==K-1, 1<=I<=N, 1<=J<=N}>
```

By merging the two regions concerning array `WORK`, we finally obtain:

```
<A(φ1)-R-MUST-{φ1==I, 1<=I<=N, 1<=J<=N}>
<WORK(φ1,φ2)-R-MUST-{φ1==J, K-1<=φ2<=K, 1<=I<=N, 1<=J<=N}>
```

3.2 Sequence of instructions

Our purpose is to calculate the regions R_B associated to the whole sequence $B = S_1, \ldots, S_n$, that is to say a summary of all the read and write references occuring in statements $S_1, \ldots, S_n$ of B.

We assume that the `READ`/`WRITE` regions of the instructions of B are known. We denote them as R_k for each statement S_k, with no distinction between `READ` and `WRITE` regions, since their propagation is identical, as we shall see later.

The region R_k is relative to the store σ_k preceding the execution of S_k. Thus, we cannot simply merge the $R_k, k \in [1..n]$ by using $\bar{\cup}$. We must first convert them into the same store. Since we know the transformer T_k corresponding to the instruction S_k, we can determine the transformation $\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}$, and hence recursively compute the region R'_k associated with the subsequence $S_k, \ldots, S_n$, by using the following equations:

$$
\begin{aligned}
R'_k &= R_k \;\bar{\cup}\; \mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(R'_{k+1}),\ \forall\, k \in [1..n-1] \\
R'_n &= R_n
\end{aligned}
\tag{1}
$$

And finally, $R_B = R'_1$.

In Equation 1, $\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(R'_{k+1})$ represents all the references read and written by instructions $S_{k+1}, \ldots, S_n$. These regions are relative to σ_k because of the use of $\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}$. Thus, it is legal to merge them with regions R_k. The result contains the elements read and written by instructions S_k and $S_{k+1}, \ldots, S_n$, which is the definition of R'_k.

As an illustration, let us consider the body of the `I` loop in our example. We are interested in the read and written references to array `WORK`. We assume that we know the regions associated to both `J` loops:

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

for the first loop, and for the second:

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

We cannot simply merge these regions because the value of `K` is modified by a call to `INC1`. The transformer that abstracts the effects of this call is `T(K)` `{K==K#init+1}`. We add its constraint to the regions corresponding to the second loop, and eliminate the variable `K` which refers to the store immediately following the call. This gives:

<WORK(ϕ_1,ϕ_2)-W-MUST-{1<=ϕ_1<=N, ϕ_2==K#init+1, 1<=I<=N}>
<WORK(ϕ_1,ϕ_2)-R-MUST-{1<=ϕ_1<=N, K#init<=ϕ_2<=K#init+1, 1<=I<=N}>

These regions are relative to the store preceding the call. Since no integer scalar variables appearing in the constraints are modified by the first `J` loop, the regions are also relative to the store preceding the execution of the first `J` loop. Thus, it is legal to merge them with the regions corresponding to the first `J` loop, after renaming `K#init` into `K`. It follows:

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

3.3 DO loop

We want to compute the regions R_L corresponding to the `DO` loop L, knowing the regions of its body. They are not only functions of the value i of loop index

I, but also of the variables modified in the loop body (collectively denoted as v). Hence, we denote them as $R(i, v)$.

These v variables are themselves functions of the loop index. We must then eliminate them 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 operator $\mathcal{T}_{\sigma_i \rightarrow \sigma_L \setminus I}$, σ_L being the store preceding the loop 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). Then the loop index is eliminated using operator $proj_I$.

The computation of the regions of a `DO` loop can finally be summarized by the following equation:

$$R_L = proj_I(\mathcal{T}_{\sigma_i \rightarrow \sigma_L \setminus I}(R(i, v)))$$

As an example, let us consider the `I` loop in Figure 2. We have seen in the previous section that the regions of the loop body concerning read accesses to the array `WORK` are:

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

They are functions of the loop index `I` and of the variable `K`, the value of which is modified in the loop body. The transformer giving the loop invariant is `T(I,K) {K==K#init+I-1}` (`K#init` is the value of `K` before the execution of the loop). We add its constraint to the previous region, and eliminate `K` (not `I`, because it is the loop index). After renaming `K#init` into `K`, it gives:

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

We then eliminate the loop index. This elimination is exact because the lower and upper bounds are affine, the value of the increment is 1, and the projection is exact in the usual meaning. We finaly obtain:

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

4 Interprocedural propagation

The interprocedural propagation of array regions is a backward analysis; the summary regions of the called subroutine are translated from the callee's name space into the caller's name space. Because of array reshaping, this operation is not straightforward.

Let us first consider the translation from a formal to an actual array. In [18], Triolet gave conditions to perform the translation by only changing the name of the array, possibly adding equalities between ϕ variables and the array reference indices at call site, and projecting the polyhedron onto the name space of the caller. These conditions are that the formal array is either similar to the actual array, or a *subarray* of the actual array, for instance a single column of a matrix.

By examining the Perfect Club Benchmarks [4], we found these conditions to be too restrictive. In particular, they cannot handle offsets in declarations, general offsets in the references at call site, and array reshaping is treated in a very

```
subroutine FOO()
dimension C(100,100,100)
call BAR(C,100,300))
```

```
<M(φ1,φ2)-IN-MUST-{1<=φ1<=n1,1<=φ2<=n2}>
   subroutine BAR(M ,n1,n2)
   dimension M(n1,n2)
```

Fig. 3. Interprocedural propagation

few cases. In order to handle the general case, we have chosen to use *subscript values* as they are described in the FORTRAN standard [1]. The subscript value of an array element is its *rank* in the array, given the fact that array elements are held in *column order*. Definition 1 expresses the fact that two corresponding elements of the actual and formal arguments must have the same subscript values, more or less the offset of the actual argument.

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

P is the calling routine, Q the called routine.

A_Q is the formal array, of dimension q, of lower and upper bounds $lq_1, \ldots, lq_q$ and $uq_1, \ldots uq_q$. The elements of A_Q are described by ϕ variables.

A_P is the corresponding actual array, of dimension p, of lower and upper bounds $lp_1, \ldots, lp_p$ and $up_1, \ldots up_p$; $op_1, \ldots, op_p$ are the indices of the reference at the call site[2]. The elements of A_P are described by ψ variables.

Definition 1. Let the *subscript values equation* be:

$$\begin{aligned}(\psi_1 - lp_1) + & \sum_{i=2}^{p}[(\psi_i - lp_i)\prod_{j=1}^{i-1}(up_j - lp_j)] - (\mathit{offset} - 1) \\ = (\phi_1 - lq_1) + & \sum_{i=2}^{q}[(\phi_i - lq_i)\prod_{j=1}^{i-1}(uq_j - lq_j)]\end{aligned} \quad (2)$$

with $\mathit{offset} = 1 + (op_1 - lp_1) + \sum_{i=2}^{p}[(op_i - lp_i)\prod_{j=1}^{i-1}(up_j - lp_j)]$. □

In the example of Figure 3, we have $A_Q = \mathtt{M}$, $q = 2$, $lq_1 = lq_2 = 1$, $uq_1 = \mathtt{n1}$, and $uq_2 = \mathtt{n2}$ for the called procedure; $A_P = \mathtt{C}$, $p = 3$, $lp_1 = lp_2 = lp_3 = 1$, $up_1 = up_2 = up_3 = 100$ for the calling subroutine; and $op_1 = op_2 = op_3 = 1$ because `CALL BAR(C, 100, 300)` is equivalent to `CALL BAR(C(1,1,1), 100, 300)`. And $\mathtt{C}(\psi_1,\psi_2,\psi_3)$ corresponds to $\mathtt{M}(\phi_1, \phi_2)$ if and only if:

$$(\psi_1 - 1) + (\psi_2 - 1) \times 100 + (\psi_3 - 1) \times 100 \times 100 = (\phi_1 - 1) + (\phi_2 - 1) \times \mathtt{n1}$$

When the subscript values equation is affine, it can be used to propagate array regions, by simply adding it to the summary region corresponding to A_Q, and eliminating ϕ variables.

Unfortunately, subcript values of arrays of two or more dimensions are seldom affine functions of ϕ or ψ variables. In our example, the equation contains $\mathtt{n1} \times \phi_2$ which is not linear. In order to decrease the degree of equation (2), we take into account the possible relations between the first common dimensions of the two arrays. For that purpose, we give two more definitions.

[2] `CALL` $Q(A_P(op_1, \ldots, op_p))$.

Definition 2. We say that arrays A_P and A_Q *have similar d-th dimension* iff:
$d = 1$ or A_P and A_Q have similar $(d-1)$-th dimension,
and $op_d = lp_d$,
and $up_d - lp_d = uq_d - lp_d$. □

Definition 3. We define the *partial subscript values equation of rank d* to be:

$$\begin{aligned} &(\psi_1 - lp_d) + \textstyle\sum_{i=d+1}^{p}[(\psi_i - lp_i)\prod_{j=d+1}^{i-1}(up_j - lp_j)] \; - (\mathit{offset}_d - 1) \\ = &(\phi_d - lq_d) + \textstyle\sum_{i=d+1}^{q}[(\phi_i - lq_i)\prod_{j=d+1}^{i-1}(uq_j - lq_j)] \end{aligned} \tag{3}$$

with $\mathit{offset}_d = 1 + (op_d - lp_d) + \sum_{i=d+1}^{p}[(op_i - lp_i)\prod_{j=d+1}^{i-1}(up_j - lp_j)]$. □

Algorithm The principle of the translation of a summary region R corresponding to A_Q into a region corresponding to A_P is the following:

1. $d = 1$
2. `while` $d \le min(p,q)$ `and` A_P `and` A_Q `have same` d`-th dimension do`
   `add` $\psi_d - lp_d = \phi_d - lq_d$ `to` R
   $d = d + 1$
3. `if` $d <= p$ `and` $d <= q$ *`(not all common dim. have been translated)`*
   `add the partial subscrit value equation (3) of rank` d `to` R
4. `else` *`(all common dimensions have been translated)`*
   `if` $d <= p$ `and` $d > q$ `(remaining dimensions of` A_P`)`
   `forall` $i \in [d..p]$ `add` $\psi_d = op_d$ `to` R
5. `forall` $i \in [1..p]$ `add` $lp_i \le \psi_i \le up_i$ `to` R
6. `eliminate` ϕ `variables from` R
7. `project the polyhedron of` R `onto the name space of` P

Step 2 consists in translating the similar common dimensions of P and Q. Two cases may appear: either one or more common dimensions have not been translated, and we must use the partial subscript values equation for the remaining dimensions (Step 3); or all common dimensions are similar, and if the actual array has more dimensions than the formal one, the remaining ψ variables are equal to the array element indices at the call site (Step 4). At Step 5, we add the bounding constraints on ψ variables. The polyhedron of R then describes the relations between ϕ and ψ variables. Steps 7 and 8 complete the translation.

At Steps 2, 3, 4, and 5, the constraints that are added to the polyhedron of R may not be affine. In that case, they are not added to the polyhedron, and the translation is considered inexact. The translated region is then a `MAY` region. Steps 6 and 7 may also yield a `MAY` region, as we have seen in Section 2. □

In our example, since `n1` $= 100$, we can add $\psi_1 = \phi_1$ to the region. Then, we build the partial subscript values equation of rank 2, which is linear:

$$(\psi_2 - 1) + (\psi_3 - 1) \times 100 = (\phi_2 - 1)$$

Step 5 then adds $\{1 \le \psi_1 \le 100, 1 \le \psi_2 \le 100, 1 \le \psi_3 \le 100\}$. Step 6 eliminates the ϕ variables. We obtain the following region:

```
<C(ψ1,ψ2,ψ3)-IN-MUST-
        {1<=ψ1<=n1, 1<=ψ2+100ψ3-100<=n2, 1<=ψ2<=100,1<=ψ3<=100}>
```

We finally substitute `n1` and `n2` by their values (respectively 100 and 300).

Regions concerning arrays in `COMMON`s are handled similarly, except that *offset* becomes the relative offset of the arrays in the `COMMON` data layout.

The previous algorithm has several advantages, especially over [5, 10]:

1. the first dimensions of the array are treated separately; even if the remaining partial subscript values equation is not affine, these dimensions are translated, and the resulting region is more precise, even if it is a `MAY` region; moreover, this leads to constraints that are simpler than with the full subscript values equation;
2. the partial subscript values equation may be affine, even if the complete subscript values equation is not linear;
3. we think that most cases that arise and that can be solved using Maslov's method [15] can also be solved by our algorithm; 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;
4. this framework is very well suited to handle arrays in `COMMON`s, because their data layout and the array bounds are known at compile time.

But we also know its limitations. It cannot solve some very frequent problems such as translating `V(N*N)` into `A(N,N)` (neither do the other methods). In a near future, we intend to extend our approach to handle more cases.

5 IN and OUT Regions

`READ` and `WRITE` regions summarize the exact effects of statements and procedures upon array elements. They do not represent the flows of array elements, which are necessary to test the legality of many optimizations. For that purpose, we introduce two new types of regions: `IN` and `OUT` regions, that take array kills [17] into account.

`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 2, 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 continuation. These are the *live* or *exported* array elements. In the program of Figure 2, 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.

The interprocedural propagation of IN regions is similar to that of READ/WRITE regions. Its a backward propagation: IN regions corresponding to a formal parameter are translated into regions corresponding to its actual parameter, 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 to form a unique summary, which is the OUT regions of the procedure. This propagation is performed using the same principles as those presented in Section 4. In the remainder of this section, we shall limit ourselves to the intraprocedural computation of IN and OUT regions for a single instruction, or a sequence of instructions.

5.1 Computation of IN regions

IN regions of an assignment are identical to the corresponding READ regions because the values of the referenced elements cannot come from the assignement itself.

We are now interested in the region IN_B corresponding to the 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 do not come from previous instructions of B.

For each instruction S_k, we denote its WRITE and IN regions as W_k and IN_k; they are relative to the store σ_k preceding S_k, and are supposed to be known.

IN'_k also represents the IN region corresponding to the subsequence $S_k, \ldots S_n$. IN_B is then defined by:

$$\begin{cases} IN_B = IN'_1 \\ IN'_k = IN_k \ \bar{\cup} \ [\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(IN'_{k+1}) \ \ominus \ W_k], \ \forall\, k \in [1..n-1] \\ IN'_n = IN_n \end{cases}$$

$\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}$ 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. $\mathcal{T}_{\sigma_{k+1} \rightarrow \sigma_k}(IN'_{k+1}) \ominus W_k$ represents the regions imported by the subsequence $S_{k+1}, \ldots, S_n$, but not defined by 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 statements of the same subsequence. This is exactly the definition of IN'_k.

As an illustration, let us consider the body of the second J loop. The READ and IN regions of its instructions concerning the array WORK are[3]:

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

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

Thus, we successively have :

$$IN'_2 = IN_2 = \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-MUST-\{}\phi_1\texttt{==J, K-1<=}\phi_2\texttt{<=K\}>}$$
$$IN'_1 = IN_1 \ \bar{\cup}\ (IN'_2 \ominus W_1) = \varnothing \ \bar{\cup}\ (IN'_2 - W_1)$$
$$= \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-MUST-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K-1\}>}$$

Finaly, $IN_B = IN'_1$ contains the sole element `WORK(J,K-1)`.

5.2 Computation of OUT regions

`OUT` regions are propagated top-down along the hierarchical control flow graph. Thus, the region OUT_B corresponding to the sequence B is supposed to be known; and we are interested in the regions OUT_k ($k \in [1..n]$) corresponding to each instruction S_k of B. These regions contain the array elements written by the current instruction, and the values of which are used by the next instructions, 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 exported outside of B. The equations defining $OUT_k, k \in [1..n]$ are then:

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

OUT_B is the `OUT` region corresponding to $B = S_1, \ldots, S_N$. It is relative to the store σ_B preceding the execution of the sequence. Therefore, $\mathcal{T}_{\sigma_B \to \sigma_n}(OUT_B)$ 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 $OUT'_n = \mathcal{T}_{\sigma_B \to \sigma_n}(OUT_B)$.

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:

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

Finaly, the `OUT` region of instruction S_k contain the elements written by S_k and exported towards the outside of B ($W_k \cap OUT'_k$), to which we must add the region written by S_k and exported towards $S_{k+1}, \ldots, S_n$ ($W_k \cap \mathcal{T}_{\sigma_{k+1} \to \sigma_k}(IN'_{k+1})$). We obtain:

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

Let us consider as an illustration the body of the second `J` loop, in Figure 2. Its `WRITE` and `IN` regions concerning the array `WORK` are the following:

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

Moreover, we assume that $OUT_B = \varnothing$. Then, we successively have:

$$
\begin{aligned}
OUT'_2 &= OUT_B = \varnothing \\
OUT_2 &= W_2 \cap OUT'_2 = \varnothing \\
OUT'1 &= OUT'_2 - W_2 = \varnothing \\
OUT_1 &= W_1 \cap (OUT'_1 \,\bar{\cup}\, IN'_2) = W_1 \cap IN'_2 \\
&= \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-MUST-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>} \\
&\cap \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-IN-MUST-\{}\phi_1\texttt{==J, K-1<=}\phi_2\texttt{<=K\}>} \\
&= \texttt{<WORK(}\phi_1\texttt{,}\phi_2\texttt{)-W-MUST-\{}\phi_1\texttt{==J, }\phi_2\texttt{==K\}>}
\end{aligned}
$$

The first instruction exports the element it defines towards the second one, which exports nothing.

6 Conclusion

We have presented the analysis of exact `READ/WRITE` array regions. They provide an effective way to exactly represent the effects of statements and procedures upon array elements. Whereas the regions initially defined by Triolet [19] are over-approximations of the effects of procedures.

Since `READ/WRITE` regions cannot be used to compute the flow of array elements, we have introduced two new types of 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.

We also provide a general linear framework for the interprocedural propagation of exact regions. It handles array reshapes, even in `COMMON`s that do not have the same data layout.

The current implementation 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 has proven the practicality of the analysis in terms of time and memory space, in spite of the exponential complexity of the operators on polyhedra.

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

Acknowledgments

We are very thankful to Corinne Ancourt, William Pugh, Fabien Coelho, Pierre Jouvelot, Alexis Platonoff and the referees for their careful reading, and helpful comments.

References

1. *American National Standard 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*, April 1991.
3. 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`.
4. M. Berry et al. The PERFECT Club benchmarks: Effective performance evaluation of supercomputers. Technical Report CSRD-827, CSRD, University of Illinois, May 1989.
5. Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 21(7):162–175, July 1986.
6. 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.
7. Fabien Coelho. Compilation of I/O communications for HPF. In *Frontiers'95*, February 1995. Available via `http://www.cri.ensmp.fr/~coelho`.
8. Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.
9. Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, September 1991.
10. Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Overview of an interprocedural automatic parallelization system. In *Fifth International Workshop on Compilers for Parallel Computers*, pages 570–579, June 1995.
11. François Irigoin. Interprocedural analyses for programming environments. In *Workshop on Environments and Tools for Parallel Scientific Computing*, September 1992.
12. François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, June 1991.
13. 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.
14. Vadim Maslov. Lazy array data-flow analysis. In *Symposium on Principles of Programming Language*, pages 311–325, January 1994.
15. 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.
16. William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
17. 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.
18. Rémi Triolet. Interprocedural analysis for program restructuring with Parafrase. Technical report 538, CSRD, University of Illinois, December 1985.
19. Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *Symposium on Compiler Construction*, 1986.

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