Array Region Analyses by Abstract Interpretation Approaches

Ilaria Lari and Laura Ricci[1]

[1] Department of Computer Science, University of Pisa
Corso Italia 40 - Pisa
ricci@di.unipi.it

Abstract—
This paper presents some static analyses to detect the portion of a data structure accessed in a region of the program, such as a loop nest. Each analysis returns a set of data access descriptors summarizing, through a set of approximations, the data accessed in the region. The analyses have been defined through an abstract interpretation framework. The paper reviews several previously defined abstract domains and shows that none of them is suitable for the analysis of interest. Then, a new domain is proposed and its integration with existing ones is defined. The integration is defined through a systematic approach to combine abstract interpretations. Using this approach, an analyzer has been automatically generated from the analysis specification. The application of the analysis to some cases of interest is shown as well.

Keywords— **data access descriptor, abstract interpretation, optimizing compiler, loop transformation, abstract domain**

I. Introduction

The problem of determining the portion of a data structure accessed in a *region* of the program, such as a loop nest or a procedure, has been recently deeply investigated. Several proposals have been presented that are characterized by the definition of an analysis computing a *data access descriptor*. The descriptor summarizes the data accessed in a region through an *approximation* of the accesses performed in the region. This kind of analysis has originally been proposed to expose *coarse grain parallelism* [B90], for instance to detect a set of loop nests or of procedure calls which may be executed in parallel, and for *array privatization* [HMALL95], to eliminate memory related dependencies. Descriptors have recently been exploited also to optimize communication patterns generated by compilers for distributed memory architectures [HPP96] and to define symbolic dependence tester.

The alternative static analyses that return access descriptors mainly differ because of the adopted approximation and, hence, they achieve distinct trade offs between the precision of the analysis and its complexity. Current approaches can be roughly classified into two groups: the first one includes proposals that bound the computational cost of the analysis by returning a coarse approximation of the accessed regions. The analysis proposed in [GS90], for instance, detects only *rectangular regions* and therefore cannot exactly describe regions accessed by *triangular* or *trapezoidal* loops: on the other hand its implementation is straightforward.

The *Regular Section Analysis* [CK88] and its extension, the *Bounded Regular Section Analysis* [HK91], are based on a symbolic analysis which returns a set of regular array accesses patterns, such as rows, columns and diagonals.

The *Data Access Descriptor* approach [B90] introduces *Simple Regions*, regions whose boundaries can be either parallel to any coordinate axis or form a 45 angle with respect to them. The data access descriptor also includes information describing how the region is traversed and how the descriptor may be redefined in the context of an enclosing region.

The approaches in the second group [TIF86, HMALL95, CI95] include analyses which return more precise results, but are characterized by a larger computational cost. These proposals summarize data access through a set of *linear restraints*. This set is returned by an analysis of the scalar values of the program. In particular, the analyses are focused on the subscripts which includes induction variables of the loops in the considered region. The first of such proposals [TIF86] is based upon the convex polyhedral theory, hence, it is rather expensive. This approach has been recently extended in [CI95] by considering a data flow analysis of the array values and by defining both *over* and *under* approximations of a region. Under approximations are exploited to compute the difference of two regions, such as in *array privatization analysis*. The *FIAT/Suif* project [HMALL95] proposes a more flexible solution through a set of operators on access descriptors. Furthermore, the operators are applied provided that they introduce no loss of information and are computationally inexpensive.

The major drawback of the proposals in the first class is that they often introduce "ad hoc" algorithms whose goal is to return information for a predefined set of applications. Furthermore, these algorithms are not fully integrated into a data flow analysis of the whole program. As a consequence, the analysis often turns out to be not general, because it can be applied only to constructs satisfying a particular set of conditions. [B90], for instance, considers loops where both the bounds of the loops and the subscripts include only the loop indexes or symbolic variables. Symbolic variables represent, for example, loop induction variables of enclosing

loops or parameters passed to a procedure.
On the other hand, the adoption of the analyses in the second class in a real world compiler is, at the moment, questionable because of their large computational complexity.
This paper presents a new analysis based upon *an abstract interpretation approach* [CC77, CH78] which integrates the data access descriptors and the data flow analysis of the whole program. The analysis does not restrict the class of loops which can be analysed. As an example, it can be applied to loops including conditional statements or to subscripts including any variable of the program. The analysis is based on the definition of a numeric domain describing the integer ranges of the variables and of a symbolic domain defining a set of linear restraints among them. With respect to other proposals, we adopt a systematic approach based on the abstract interpretation framework, which reduce the complexity of the analysis and enables the automatic generation of an analyzer implementing the analysis. The complexity of the analysis is reduced because of a *modular definition* of the domains. The numerical properties of the variables, rs. their relations, are described by different domains: the connection between the domains is considered only in the definition of their reduced product and of the abstract operators. The analysis has been implemented through *PAG*[AM95], a tool for the automatic generation of analyzers from abstract interpretation based specifications.

[B95] proposes a *symbolic dependence tester* based on an analysis which presents some similarities with our approach. A single abstract structure, the *symbolic range*, describes both the numerical properties and the relations among the variables: this increases the complexity of the analysis. Furthermore, neither the abstract domain definition nor the correctness proofs are presented. We recall that the definition of an abstract domain is essential to exploit automatic analyzer generation tools. Finally, the definition of the abstract operators is based on the manipulation and the simplification of arbitrary complex symbolic expressions [HP96]. On the other hand, our analysis computes *integer ranges* for the variables, and a simple symbolic calculus suffices to relate the symbolic domain and the numeric one.

First, the paper reviews a set of abstract interpretation domains proposed in literature, and shows that none of them is suitable for our purposes. Then, a new abstract domain is proposed and its combination with existing domains is defined: the combination is specified according to the general methodology described in [CC77].

II. The Abstract Interpretation Approach

An abstract interpretation approximates program properties by evaluating a program on a simpler, and computer-representable, domain of descriptions of "concrete" program states. This implies the definition of an approximate(abstract) semantics defined on an *abstract domain* that exhibits a structure, i.e. *an ordering*, which is somehow present in the richer structure associated to program execution. To define an abstract interpretation, a relation between the concrete and the abstract domains of the program has to be defined. According to the Cousots approach, this relation is defined through a pair of functions, the *abstraction function* α and the *concretization function* γ, that define a *Galois Connection* [CC77]. The definition of a *Galois Connection* guarantees that both α and γ are monotonic, that α introduces an approximation, while the concretization function γ does not introduce any loss of information. When the Galois Connections between the concrete and the abstract domain has been defined, any concrete fixpoint can be approximated by an abstract fixpoint through an approximation of the concrete semantics function. Thus, an abstract interpretation framework is characterized by an abstract domain which has a Galois Insertion with the concrete semantic domain and by an *Approximate Semantic Function*. This function introduces, for each instruction, an *abstract operator* that defines its behavior on the abstract domain.

The framework also requires that the abstract semantics satisfies a *correctness condition*: each state produced by applying an instruction i to a concrete state c is approximated by the abstract state which is returned by the abstract operator that corresponds to i when it is applied to the abstraction (obtained by the application of α) of c.

Finally, the termination of the analysis is guaranteed by considering some *finiteness properties* of the abstract domain. The abstract domain may be finite, or it may satisfy the *ascending chain condition*, which states that ascending chains are bounded. If the abstract domain does not satisfy any of these conditions, an infinite number of steps may be required to ensure the convergence to the least fixpoint. It may also be the case that the number of steps is not infinite, but is so large that the analysis becomes impractical. In this case alternative techniques have to be considered to speed up the fixpoint computation: for instance a "widening/narrowing" operator may be defined.

III. Analysis of Existing Domains

This section evaluates the feasibility of exploiting existing domains. We will assume that a set of auxiliary variables is introduced into the program to be analysed where the value of each of these variables is a function of array subscripts. We have to introduce, at least, the assignments of the values of the subscripts to a set of auxiliary variables. Each analysis returns some properties of auxiliary variables that depend on the adopted abstract domains. The regions accessed by a reference can be deduced through the analysis of these prop-

erties. As noted in [B90], a larger set of program properties may be deduced through further auxiliary variables whose value may be, for instance, the sum and the difference of the subscripts.

A. The Domain of Intervals

An analysis to detect the interval of values that each variable may assume in each point of a program has been defined in [CC77]. This domain may be exploited to describe any kind of array data accesses, but, it often returns a coarse approximation for triangular or trapezoidal loops.

Nevertheless, as we will show in section V, better results may be achieved when the analysis is combined with a *relational analysis*, i.e. an analysis able to detect some relations among the variables of the program.

Figure 1 shows the abstract domain of intervals, Int defined for a program including only one variable. In this case, the concrete domain is $(\mathcal{P}(\mathbf{Z}), \subset)$, while the abstract domain includes the set of all integer intervals; the abstraction function $\alpha(S)$, $S \in \mathcal{P}(\mathbf{Z})$ is defined as follows

$$\alpha(S) = \begin{bmatrix} \text{Min(x)} & \text{,Max(y)} \\ x \in S & y \in S \end{bmatrix}$$

while the concretization function is defined as follows:

$$\gamma([a,b]) = \{x | (x \in Z) \wedge (a \leq x \leq b)\}$$

The abstract operators are defined according to the interval arithmetic. As an example, our analysis is defined in terms of the following operators on intervals:

$$\begin{array}{lcl} [a,b] + [c,d] & = & [a+c, b+d] \\ [a,b] - [c,d] & = & [a-d, b-c] \\ [a,b] * [c,d] & = & [min(ac,ad,bc,bd), max(ac,ad,bc,bd)] \end{array}$$

Since the abstract domain is infinite, a *widening* ∇ and a *narrowing* Δ is defined; in the following we will exploit the definition given in [CC77].

Given a reference to an array A, the analysis computes an interval of values for each subscript in the reference and for the sum and the difference of the references. These intervals define the region accessed by an array reference. The analysis cannot capture the relation between the induction variables. Hence, as shown in the following example, it may return a rectangular region even if the program accesses a triangular or trapezoidal region.

Example III.1 *Consider the loop:*

```
for I :=1 to 10 do
    for J :=1 to 15-I do
        A[I, J]:=...
```

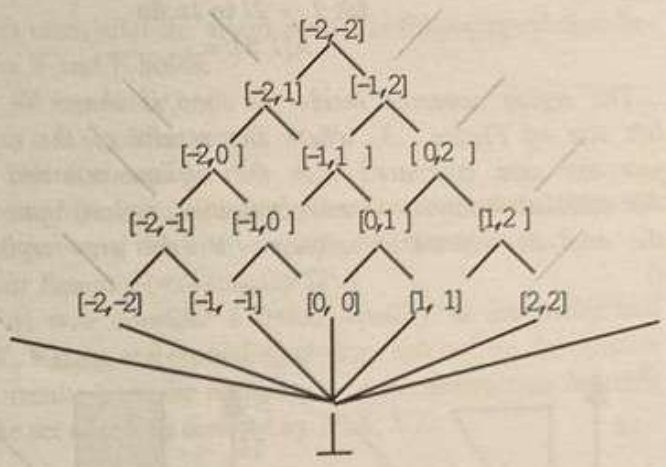


Fig. 1. The Interval Domain

the program is initially modified by introducing the auxiliary variables S_1,S_2,S_3,S_4 that are equal to, respectively, the value of the two subscripts, their sum, and their difference. The region returned by the analysis is the left one in Fig 2, while the right one describes the exact region accessed by the loop. Notice that the analysis returns a coarse approximation, because it does not exploit the relationship between the values of J and I due to the upper bound of the innermost loop. ◇

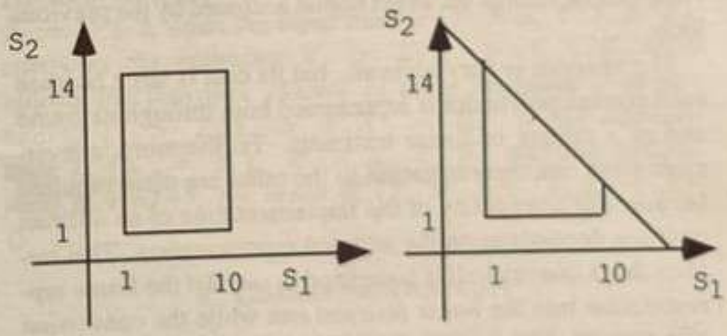


Fig. 2. Region returned by the interval analysis

B. The Domain of Linear Restraint

As discussed in [B90] the bounds of the region accessed in a loop are often parallel to the axes or to the bisecting lines. Nevertheless, the following example shows that, in some cases, the region is more complex:

Example III.2 *Consider the loop:*

for *I* := 2 **to** 8 **do**
  for *J* := 2*I* **to** 16 **do**
    A[I, J]:=...

The region accessed inside the loop is shown in the left size of Figure 3, where the results of the analysis are also compared with the regions returned by the interval analysis (the rectangular region) and by the analysis defined in section V (the grey region). ◊

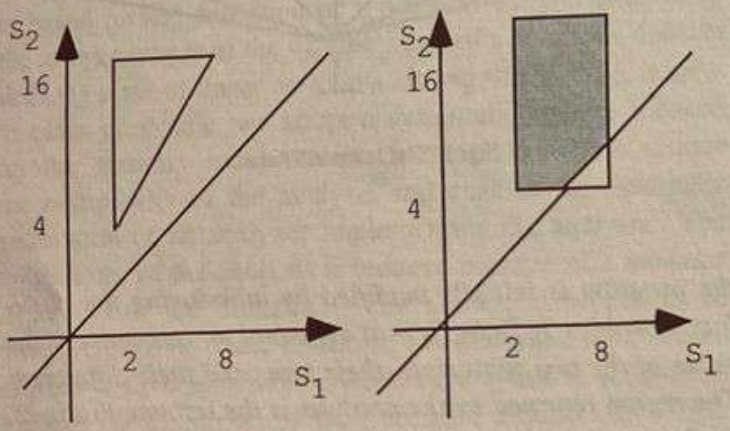


Fig. 3. Regions Returned by Linear Restraints and Int * Bisectors

The analysis based on the *linear restraint domain* [CH78] can describe exactly this kind of regions because it returns a set of linear restraints among the variables of the program. Hence it can returns the exact region accessed by the previous loop.

The analysis is very accurate, but its cost is large because each convex polyhedra is represented both through its frame and as a system of linear restraints. Furthermore, conversions from one representation to the other are often required because the complexity of the implementation of an abstract operator depends upon the adopted representation. This implies that a convex hull is computed to convert the frame representation into the linear restraint one while the conversion of a system of linear restraints into a frame requires the application of the pivot method.

IV. The Domain of Bisectors

A detailed examination of current domains shows that, while the analysis of interest requires a relational analysis, the linear restraint domain is too complex for this purpose.

Hence, we introduce a new domain by defining a set of simple restraints and by integrating this domain with that of the intervals. The implementation of the operators of interest is straightforward.

Let us consider a program that uses only two variables. In this case, the domain includes linear restraints which describe one of the regions that results by cutting the plane through bisecting lines: for this reason we call this domain the *Bisectors Domain*.

For the sake of conciseness, here we will show only one subset of the domain, including the restraints corresponding to the bisecting line $X = Y$. The domain is shown in Figure 4, where $X?Y$ means that there may be any kind of relationship between X and Y.

The full specification of the *Bisectors Domain* is given in [L98] and it includes the restraints corresponding to the bisecting line $X = -Y$ as well. The specification has been partitioned by defining the domain corresponding to the two bisectors separately. Then the domains are combined by defining their *reduced product*. We will describe this methodology in Section V.

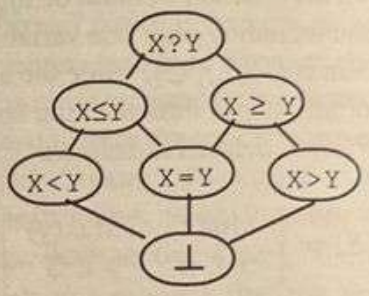


Fig. 4. The domain of bisectors

Since the analysis returns only equality, less than or greater than relations between the variables, it is obviously less expressive than the linear restraint one. However its combination with the analysis of intervals increases its expressiveness and it can return meaningful information in several real cases.

Before defining the domain, we stress that the domain by itself cannot be exploited for region analysis and that this definition has to be considered only as a first step to define the combined analysis of section V.

Since we assume that the program includes two variables whose values belong to Z, the concrete domain is defined as $(\mathcal{P}(\mathbf{Z}\mathbf{x}\mathbf{Z}), \subseteq)$ The definition of the abstraction function is the following:

Definition IV.1 *Let us consider a set* $A \in \mathcal{P}(\mathbf{Z}\mathbf{x}\mathbf{Z})$. *The abstraction function* $\alpha : \mathcal{P}(\mathbf{Z}\mathbf{x}\mathbf{Z}) \rightarrow Bisector$, *is defined as follows:*

$$\alpha(A) = \begin{cases} \bot & \text{if } A = \emptyset \\ X = Y & \text{if } \forall (i,j) \in A,\ i{=}j \\ X > Y & \text{if } \forall (i,j) \in A,\ i{>}j \\ X \geq Y & \text{if } \forall (i,j) \in A,\ i{>}j \vee i{=}j \\ X < Y & \text{if } \forall (i,j) \in A,\ i{<}j \\ X \leq Y & \text{if } \forall (i,j) \in A,\ i{<}j \vee i{=}j \\ X?Y & \text{otherwise} \end{cases}$$

◊

We define now the concretization function:

Definition IV.2 *Let us consider a pair of variables so that $XrY \in Bisector$; the concretization function $\gamma : Bisector \rightarrow \mathcal{P}(\mathbf{Z}\times\mathbf{Z})$ is defined as follows*

$$\gamma(XrY) = \begin{cases} \{(i,j)\ t.c.\ i \in \mathbf{Z}, j \in \mathbf{Z},\ i = j\} & if\ X = Y \\ \{(i,j)\ t.c.\ i \in \mathbf{Z}, j \in \mathbf{Z},\ i > j\} & if\ X > Y \\ \{(i,j)\ t.c.\ i \in \mathbf{Z}, j \in \mathbf{Z},\ i \geq j\} & if\ X \geq Y \\ \{(i,j)\ t.c.\ i \in \mathbf{Z}, j \in \mathbf{Z},\ i < j\} & if\ X < Y \\ \{(i,j)\ t.c.\ i \in \mathbf{Z}, j \in \mathbf{Z},\ i \leq j\} & if\ X \leq Y \\ \mathbf{Z}\times\mathbf{Z} & if\ X?Y \\ \emptyset & if\ \perp \end{cases}$$

◊

The two functions define a *Galois Connection*[L98].

The kernel of the analysis is the definition of an abstract operator for each instruction of the program.

Invertible Assignments

First we consider the abstract operator corresponding to *invertible assignments*, i.e. assignments where the new value of the assigned variable is a function of its old value. The most general form of an invertible assignment(we consider linear expressions only) is:

$$X := aX + bY + c$$

where a, b, c are integer constants.

In the most general case, no information can be returned by the abstract operator because the relation between X and Y after the assignment depends upon both the relation between them before the assignment and their signs. This information will be available when we combine this domain with the intervals one. For now, we can notice that the abstract invertible assignment operator returns meaningful information only if the assignment has the following structure:

$$X := X + c$$

In this case, the relationship existing between X and Y after an assignment A is deduced from the relationship between them before A by considering the sign of c as follows:

- if c is *positive* we have that:

relation before A	*relation after A*
$X = Y$	$X > Y$
$X > Y$ or $X \geq Y$	$X > Y$
$X < Y$ or $X \leq Y$	$X?Y$
$X?Y$	X?Y

- the case where c is *negative* is analogous;
- if c=0 the relation does not change.

Non invertible assignment

Let us consider now a non-invertible assignment, that is an assignment of the form:

$$X := aY + b$$

Also in this case, the relationship between X and Y after the assignment depends on their sign: meaningful information can be obtained only in the following case :

$$X := Y + b$$

In this case, after the assignment, the following relation between X and Y holds:

- *if b is positive*, then $X > Y$
- *if b is negative*, then $X < Y$
- *if b=0*, then $X = Y$

Meaningful information for further cases is deduced from the combination of this analysis with the interval one.

Linear Equality or Inequality Tests

Let us now consider a linear equality or inequality test $P(X,Y)$; the corresponding abstract test returns the relation that results from the application of the abstraction function to the set of values denoted by $P(X,Y)$.

Example IV.1 *Consider the test $X \leq Y + b$ where b is positive, the abstract operator returns $X?Y$. If the test is $X = Y + b$, where b is positive, $X > Y$ is returned* ◊

The abstract domain has been extended to an arbitrary number of variables, but, in order to limit its complexity, the resulting analysis only returns restraints between *pairs of variables*. Hence, the elements of the extended domain are a set of restraints, where each restraint belongs to the bisector domain and it involves only one pair of variables.

The abstract operators may be easily deduced from those presented in the previous sections by considering the transitive closure of the restraints returned by the operators.

Example IV.2

The abstract interpretation of the non-invertible assignment $X := Y + 2$, when the input state is $\{Y > Z, X?Y, X?Z\}$ computes the restraints $Y > Z, X > Y, X?Z$, then returns the transitive closure of these restraints thus obtaining $Y > Z, X > Y, X > Z$. If the abstract state is $\{X = Y, Y < Z, X < Z\}$ and the assignment is Y:=Y+5, the result is $\{Y > X, Y?Z, X < Z\}$

◊

V. Combining Domains

This section shows how more powerful analyses are defined by combining the domains in the previous sections.

A. The Direct Product Int × Bisectors

[CC79] proposes a set of methodologies to combine domains. The simplest one is the *Direct Product* approach where a pair of analyses A_1, A_2 are combined by defining a domain that is the cartesian product of those of, respectively, A_1 and A_2. The abstraction returns the pair of abstract values a_1, a_2 by applying, respectively, the abstraction function of A_1 and of A_2. The concretization function returns the intersection of the values obtained by the concretization of the pair a_1, a_2. Each abstract operator returns the pair obtained

by separately applying the corresponding abstract operators defined by A_1, rs. A_2 to a_1, rs. a_2 Even if this approach is very simple and does not require any change in the specification of the original analysis, it does not define an optimal analysis. Actually, the domain $Int \times Bisector$ returns a more precise approximation for triangular or trapezoidal loops, but it does not exploit all the information obtained from the combination of the domains. This is shown in the following example.

Example V.1

Let's consider again the program defined in example III.2. The bisector analysis cannot detect that $j > i$ when the array is accessed, because this relationship depends on the sign of i. Hence, the same result of the interval analysis is returned ◊

*B. The Reduced Product Int * Bisectors*

A more refined approach combines the analyses through their *Reduced Product*. This approach defines a *reduction operator*: given an element (a_1, a_2) belonging to the cartesian product of the domains of the original analyses, the reduction operator may refine both a_1 and a_2 by considering the intersection of the concretizations of a_1, rs. a_2 and then by reabstracting the result through the corresponding domains. The reduced domain is produced by reducing each element in the cartesian product of the original domains.

To specify on optimal analysis we have to redefine the abstract operators to exploit the information conveyed by each domain.

In the following, we show the reduction and the abstract operator with respect to the simplified bisector domain shown in section 4.

The reduction operator

$$\sigma : Bisector \times Int \times Int \rightarrow Bisector \times Int \times Int$$

is applied to a triple $(R_{x,y}, I_x, I_y)$, where $I_x = [m_x, M_x]$, $I_y = [m_y, M_y]$. This operator may return a more precise relation or it may restrict the bounds of the intervals. We show only some cases of reduction: let $m = max(m_x, m_y)$, $M = min(M_x, M_y)$

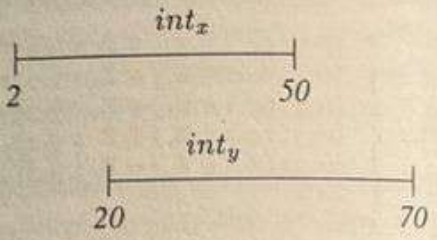
$$\begin{array}{lll}
\sigma(X = Y, I_x, I_y) = & \bot, \bot, \bot & \text{if } (I_x \cap I_y) = \emptyset \\
\sigma(X = Y, I_x, I_y) = & X = Y, [m, M], [m, M] & \text{if } (I_x \cap I_y) \neq \emptyset \\
\sigma(X > Y, I_x, I_y) = & \bot, \bot, \bot & \text{if } M_x \leq m_y \\
\sigma(X \geq Y, I_x, I_y) = & \bot, \bot, \bot & \text{if } M_x < m_y \\
\sigma(X \geq Y, I_x, I_y) = & X \geq Y, [m, M_x], [m_y, M] & \text{if } (I_x \cap I_y) \neq \emptyset \\
\sigma(X ? Y, I_x, I_y) = & X > Y, I_x, I_y & \text{if } m_x > M_y \\
& \ldots &
\end{array}$$

The values returned by the operator when $X < Y$ or $X \leq Y$ can be trivially deduced from the previous cases.

Example V.2 *Consider the reduction of the state*

$$\sigma(X > Y, [2, 50], [20, 70])$$

the interval values of X and Y are sketched in the following picture:



Since $X > Y$, all the values of X belonging to $[2, 20]$ can be discarded, because any value in this interval is not larger than any value of Y. Furthermore, the values of Y belonging to $[50, 70]$ can be discarded as well, because any value of Y in this interval is not lower than any value of X. Hence,the reduction operator returns the triple

$$\sigma(X > Y, [2, 50], [20, 70]) = X > Y, [21, 50], [20, 49]$$

Let us now consider the reduction of the triple:

$$(X \geq Y, [17, 20], [3, 15])$$

Since the intervals do not intersect, any value of X is larger than any value of Y. The reduction operator may now return a tighter relation between the variables X and Y as follows

$$\sigma(X \geq Y, [17, 20], [3, 15]) = (X > Y, [17, 20], [3, 15])$$

◊

As far as concerns the abstract operators, we will show only one case of invertible assignment; the full analysis is given in [L98]. Let us consider the assignment

$$X := X - Y$$

The corresponding abstract operator A is applied to an abstract state, (R_{xy}, I_x, I_y), where $I_x = [m_x, M_x]$, $I_y = [m_y, M_y]$ and R_{xy} is the relation between X and Y. The operator returns a triple whose elements describe, respectively, the relation between X and Y and the intervals of the values of X and of Y after the assignment. Some cases returned by the operator are the following:

$$\begin{array}{lll}
A(X > Y, I_x, I_y) = & > (I_x - I_y) \cap [1, +\infty), I_y & \text{if } M_y \leq 0 \\
A(X > Y, I_x, I_y) = & ?, (I_x - I_y) \cap [1, +\infty), I_y & \text{if } m_y > 0 \\
A(X \geq Y, I_x, I_y) = & \geq, (I_x - I_y) \cap [0, +\infty), I_y & \text{if } M_y < 0 \\
A(X \geq Y, I_x, I_y) = & \geq, (I_x - I_y) \cap [0, +\infty), I_y & \text{if } M_y \leq 0 \\
A(X = Y, I_x, I_y) = & \leq, [0, 0], I_y & \text{if } m_y \geq 0 \\
A(X = Y, I_x, I_y) = & \geq, [0, 0], I_y & \text{if } M_y \leq 0 \\
\ldots & &
\end{array}$$

The other cases may be easily deduced.

Example V.3 *Let us consider the abstract state:*

$$X > Y, I_x = [-13, -10], I_y = [-20, -2]$$

it can be tightened by the reduction operator as follows:

$$X > Y, [-13, -10], I_y = [-20, -11]$$

When applied to the previous triple, the abstract invertible assignment exploits the relation existing between X and Y, to deduce that the new value of X will be positive. Hence, the new interval of X is computed as follows:

$$Inew_x = (I_x - I_y) \cap [1, +\infty) = \\ ([-13, -10] - [-20, -9]) \cap [1, +\infty) = \\ [-2, 10] \cap [1, +\infty) = [1, 10],$$

while the interval for Y is not modified. The relation $X > Y$ is preserved by the assignment, because the value of Y is negative. Hence, the operator returns the triple:

$$X > Y, [1, 10], I_y = [-20, -9] \qquad \Diamond$$

Example V.4 *Let us consider again the example III.2: now the analysis is able to deduce that, in any access, $j > i$, by exploiting the positiveness of i. Furthermore the analysis deduces that $2 \leq i \leq 8$, $4 \leq j \leq 16$. The concretization of these restraints returns their intersection. The interval analysis returns the rectangular region shown in the right size of Fig 3; the $Int * Bisectors$ analysis return the grey region. We can notice that the exact region is defined by a bound that is not parallel to a bisecting lines: this bound is approximated by the one $S_2 > S_1$.* $\Diamond$

VI. Applications

This section shows that the analysis can to deduce information that can be exploited in the parallelization of real programs even if the exact range of values of the subscripts cannot be deduced at compile time.

Consider the following program

```
for I := 6 to 10 do
    for J := I + 1 to 11 do
        i1 :  A[I + J, I − J] := ...
        i2 :  A[I + J, J − I] := ...
```

At first the program is transformed by introducing a set of auxiliary variables recording the values of the subscripts, their sum and their difference. Consider now the direct product $Int \times Bisectors$: it returns an approximation of the regions accessed by i_1 and i_2. The region for i_1 is the region R_{i1} shown in the upper part of Figure 5 plus the triangle symmetrical to R_{i1} with respect to the axis CD. The region returned for i_2 is the region R_{i2} plus the triangle symmetrical to R_{i2} with respect to the axis AB. Since the two regions are not disjoint, no transformation is possible

Consider now the domain $Int * Bisectors$: the corresponding analysis returns an exact result, i.e. i_1 accesses R_{i1}, while i_2 accesses R_{i2}. Note that, in R_{i1} and R_{i2}, the bounds parallel to the bisecting lines are obtained by considering the intervals returned by the analysis for the sum and the difference of the subscripts. In this case, the analysis does not introduce any approximation. By applying the operators on the reduced domain, the analysis detects that $J > I$ whenever the loop body is evaluated, and it can exploit this information when evaluating the subscripts. In particular, while the value of the subscript $I - J$ is negative, the value of $J - I$ is always positive. Since the regions accessed by the references i_1 and i_2 are disjoint, the two accesses may be performed in parallel. Several transformations may be applied to the previous loop according to the desired granularity. As an example, the loop may be transformed as follows:

```
parsection
for I := 6 to 10 do
    for J := I + 1 to 11 do
        i1 A[I + J, I − J] := ...
for I := 6 to 10 do
    for J := I + 1 to 11 do
        i2 A[I + J, J − I] := ...
endparsection
```

A finer grain of parallelism may be achieved by executing the iterations of the outermost loop sequentially and the iteration of innermost ones in parallel.

Let us suppose now that the previous program has been modified as follows:

```
forI := 6 to 10 do
    for J := I + 1 to 11 do
        i'1 :  A[I + J, I − J − C] := ...
        i'2 :  A[I + J, J − I + C] := ...
```

Suppose that the analysis $Int * Bisectors$ detects that $C \in [0, 2]$ each time the loop is executed: in this case i'_1, rs. i'_2 access the regions $R_{i'_1}$, rs. $R_{i'_2}$ shown in the lower part of Fig. 5. Even if each region is now an approximation of the exact region referred by each instruction, the analysis can detect that the regions are disjoint.

This example shows that our approach naturally integrates the array region analysis with a data flow analysis of the whole program and it supports the analysis of a richer set of programs with respect to most of existing approaches.

VII. Conclusions

The proposal presented in this paper is the kernel of the data accesses analysis we have defined for the parallelizing

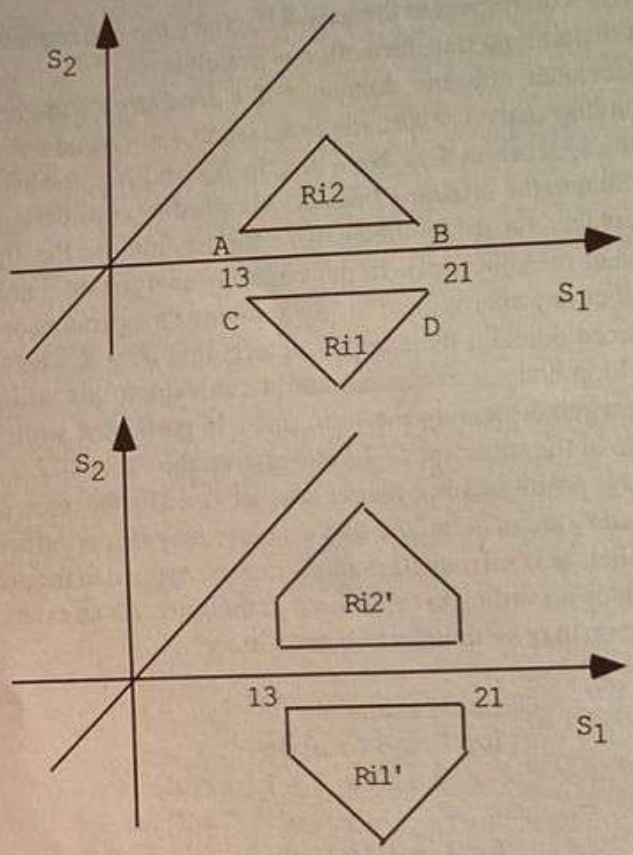


Fig. 5. Regions Computed by the analysis

compiler developed within the *PQE2000* [V98] project to detect *coarse grain* parallelism. The whole specification of the analysis requires about one hundred lines of *PAG* specification code. We are now testing the analysis on a set of significative benchmarks. The first results show that the approximation returned by the analysis convey useful information mainly if the array references include *coupled subscripts*. In this case the analyzer can exploit the relations existing among the variables in the subscript. We plan to extend the analysis both by refining the domains and by defining their composition with other abstract domains. For instance, we are going to extend the bisector domain to include bounds parallel to the bisecting lines and we will exploit arithmetic congruences to describe non convex regions.

References

[AM95] M.Alt, F.Martin,"Generation of efficient interprocedural analyzers with PAG", *Static Analysis Symposium, SAS 95*, LNCS 983,1995, pp. 33-50, Springer Verlag .

[B90] V.Balasundaram,"A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: The Data Access Descriptor", *Journal of Parallel and Distributed Computing*, 1990, vol 9, pp 154-170.

[B95] W.J.Blume,"Symbolic Analysis Techniques for Effective Automatic Parallelization", PhD thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing, June 1995.

[CK88] D.Callahan,K.Kennedy,"Analysis of Interprocedural Side Effects in a Parallel Programming Environment", *Journal of Parallel and Distributed Computing*, 1988, vol 5, pp 517-550.

[CC77] P.Cousot,R.Cousot,"Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoint",*Fourth ACM symposium on Principles of Programming Languages*,pp 238-252, 1977.

[CC79] P.Cousot,R.Cousot,"Systematic Design of program Analysis Framework", *Sixth ACM symposium on Principles of Programming Languages*,pp 269-282, 1979.

[CH78] P.Cousot,N.Halbwachs"Automatic Discovery of Linear Restraints among Variables of a Program", *Fifth ACM Symposium on Principles of Programming Languages*,pp 84-96, 1978.

[CI95] B.Creusillet,F.Irigoin,"Interprocedural Array Region Analyses", *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp 46-60, August 1995.

[GS90] T.Gross, P.Steenkiste,"Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compiler", *Software- Practice and Experience*, February 1990, vol 20(2), pp 133-155.

[HP96] M.Haghighat, C.Polychronoupoulos,"Symbolic Analysis for Parallelizing Compilers " *ACM Transactions on Programming Language and Systems*, 1996, Vol.18, No. 4, July 1996.

[HMALL95] M.W.Hall, B.R.Murphy e S.P.Amarasinghe, S.Liao, M.S.Lam,"Interprocedural Parallization Analysis: a case study", *Proceedings of 8th International Workshop on Languages and Compilers for parallel Computing*,August 1995.

[HK91] P.Havlak, K. Kennedy,"An Implementation of Interprocedural Bounded Regular Section Analysis", *IEEE Transaction on Parallel and Distributed System*, July 1991,No. 3, Vol.2.

[HPP96] J. Hoeflinger,Y. Paek, D.Padua,"Region-based Parallelization Using the Region Test " Technical Report 1514, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing , December 1996.

[L98] I.Lari,"Data Access Descriptor Computation: Abstract Interpretation Proposals",*Master Thesis* Department of Computer Science, University of Pisa.

[TIF86] R.Triolet, F.Irigoin, P.Feautrier,"Direct Parallelization of Call Statement", University of Illinois, Center of Supercomputing R & D, USA, 1986

[V98] M. Vanneschi,"Heterogeneous HPC Environments", *Euro-Par 98*, LNCS 14707, pages 21-34, Southampton, UK, September 1998.