Alias Analysis for Fortran90 Array Slices

K. Gopinath & R. Seshadri*
Department of Computer Science & Automation
Indian Institute of Science, Bangalore

Abstract
Most alias analyses produce approximate results in the presence of array slices. This may lead to inefficient code which is of concern, especially, in languages like Fortran90. In this paper, we present an overview of a static alias analysis that gives accurate results in the presence of array slices in Fortran90.

1 Introduction

An optimising compiler requires accurate data-flow analysis to perform code-improving transformations and generate efficient code. To perform effective data-flow analysis, we require precise dependence and alias information. An alias occurs at some program point during program execution when 2 or more names exist for the same memory location, ie. when two or more *l-value* expressions refer to the same memory location. The alias problem is to determine the alias relationships at each program point. Alias analysis has been initially developed for non-pointer, non-recursive languages like Fortran77. Recent interest has shifted to languages like C that support pointers.

Obtaining accurate alias information for pointer supporting languages becomes difficult because of the following: the address-of operator can lead to the creation of a new pointer relationship at any program point; passing pointers to functions can lead to the called function modifying the alias relationship of the callee; dynamic allocation of memory for pointers; and recursive structures and functions could lead to unbounded number of alias relations.

In addition, an accurate alias analysis needs the context in which the function is called. Various approaches to this problem are by restricting the flow of alias information to *realizable paths* only[8], by using *source alias sets* of the last call site [9] or by using *invocation graphs* [7].

Fortran90 has a restricted notion of pointers compared to C. But it has support for pointers to array slices which are very useful for specifying computation in SIMD, MIMD and other similar machines. The existing alias algorithms for pointer-based languages provide approximate information about aliases to array slices as they do not make any distinction between references to different locations in the same array. Emami[7] does make a distinction between the alias to the first element of the array and the alias to the rest of the array but this is still not enough to *exploit* the parallelism available in programs that use array slices.

We have extended the alias algorithm reported in [7] to produce more accurate results in the presence of array indices for Fortran90. Our algorithm assumes a SSA form of representation in which every redefinition of a variable is renamed.

2 Motivation

Consider the following example from Fortran90:

```
SUBROUTINE gauss_elim()
   REAL dimension(10,10),target :: A
   INTEGER i,n,j,maxrowloc
   INTEGER dimension(:),pointer::x,y,z
   INTEGER dimension(:),pointer::temprow,maxrow

S1: DO 50, i = 1,n
S2:    x => A(i,i:n)
S3:    z => A(i:n,i)
S4:    maxrowloc = MAXLOC(z)
S5:    maxrow => A(maxrowloc,i:n)
S6:    ALLOCATE(temprow(n-i))
       IF (maxrowloc.ne.i)
         temprow = x
         x = maxrow
         maxrow = temprow
       ENDIF
S7:    DO 200 j = i+1, n
S8:      y => A(j,i:n)
S9:      y = y - x*(y(1)/x(1))
```

*Author for correspondence:gopi@csa.iisc.ernet.in

```
S10:  200 ENDDO
        ...
S11:50 ENDDO
END
```

In Fortran90, pointer assignments are distinguished from normal assignments by =>. At S2, there is a pointer assignment with x pointing to a single dimensional array subsection $A(i, i:n)$. This results in the alias pair $< *x, A(i, i:n) >$. x points to the i^{th} row starting from the i^{th} element. z points to the i^{th} column starting from the i^{th} element.

The function MAXLOC returns the index of the maximum valued element. The statements between S6 and S7 swap the elements of the i^{th} row and the maxrow. Note that x still points to the i^{th} row. Statements S7 to S10 perform row operations which transform the matrix to upper triangular matrix. Note that i and j do not take the same value; this can be determined by a very simple intraprocedural range analysis using reaching definitions. Hence, the array subsection pointed to by x and y never overlap. Therefore, at S8, we have the alias pairs $\{< *y, A(j, i:n) >, < *x, A(i, i:n) >\}$. We can move the *normalizing operation* $x/x(1)$ outside the loop and rewrite statements S7 to S11 based on the alias information as:

```
S7':  normtemp = x/x(1)
      DO 200, j = i+1,n
         y => A(j,i:n)
         y = y - normtemp * y(1)
S11': 200 ENDDO
```

Note that it is not possible to perform this optimization if it is not known if x and y refer to the same slice. If they refer to the same slice or if there is some overlap in the slices they refer to, incorrect results will be obtained. More interestingly, this optimization cannot be detected easily (or if at all) if the slice notation is not used. We discuss the results of our analysis for this problem in Section 3.

Most of the parallelism in programs is present in loops. An accurate information about the induction variables and the alias information will help in code optimisations. Wolfe[12] detects common forms of induction variables using SSA form. Consider the example of *flip-flop variables*:

```
 j = 1
 jold = 2
 DO iter = 1,n  /* relaxation code */
  x => A(jold,*,*)
  y => A(j,*,*)
  DO k = 2,n
   y(k) = y(k-1) - x(2)/x(1,1)
  ENDDO
  temp = jold
  jold = j
  j = temp
 ENDDO
```

j and jold never take the same value but take either 1 or 2. Hence, x and y do not point to overlapping areas and we get the alias pairs as $\{< *x, A(j, *, *) >, < *y, A(jold, *, *) >\}$. Hence, we can move the computation $x(2)/x(1,1)$ out the loop as it is a loop invariant.

Another case that SSA can help is locating *monotonically increasing variables:* this can be used to show that one array slice can never be aliased by another. Our alias framework also needs SSA, in addition to its role in detecting induction variables.

Overview of Alias Analysis Framework

We use the basic framework in [7] for interprocedural alias analysis but whose implementation targeted the aliasing problem in the context of a parallelizing, optimizing C compiler (McCat). The salient features of this approach[7] are:

1: Instead of computing alias pairs, a *points-to* analysis is performed. The points-to abstraction abstracts the set of accessible real stack locations with a finite set of named abstract locations on an abstract stack. Every real stack location (source or target) is represented by exactly 1 named abstract location whereas each abstract location may represent 1 or more real stack locations. While an abstract stack location corresponds to a local, global or parameter variable, only a variable that is part of some alias relationship has an entry in in the abstract stack. The advantage of this representation is that both *may* and *must* can be easily derived. The points-to abstraction is a *compact* representation of the alias information and is equivalent to the way Choi et al [4] represent aliases.

2: The analysis computes both *definite* and *possible* points-to relationships simultaneously. A *definite* relationship implies that on all execution paths the relationship holds while a *possible* relationship may hold on some execution path. Definite relationships provide killing information which can be used to improve accuracy.

3: Separate methods of alias analysis are provided between stack variables and between references to dynamically-allocated objects on the heap.

4: Interprocedural analysis is performed on a modified call graph (the *invocation graph*) which encodes all possible interprocedural realizable function call sequences. The invocation graph explicitly represents all possible invocation paths with each invocation chain having a unique path in the graph, thus providing the calling context. The invocation graph, for non-recursive programs, is built by a depth-first traversal of the call graph. Approximate nodes are added to the graph to prevent infinite unrolling if recursion is present.

5: A simple compositional analysis is used to compute points-to relationships. Rules are given for the analysis of basic constructs in the program (simple statements, if, while) and these rules are used to compute the compound constructs (sequencing, function calls). Recursion requires a fixed point computation.

Overview of the Problem

In [7], the set of accessible real stack locations is abstracted with a finite set of named abstract locations on an abstract stack. With array slices, we need to ensure that there can be only one abstract location for a given data structure. We cannot have 2 abstract locations like $A[j:7]$ and $A[1:6]$ that may overlap.

Subscripted variables with the same base array but non-overlapping indices (this includes various slices present in the program) must have unique entries in the abstract stack for accurate alias analysis. For example, array locations $A[10], A[23], A[i]$ have different locations in the abstract stack if i can never assume 10 or 23 as values. If i can assume values in the range, say, 8..14, then there will be 2 abstract stack locations: $A[8..14]$ and $A[23]$.

One solution is to do analysis on the indices. But it is also important to find ways of decomposing the data structure (in this case arrays) so that each slice can be seen to be independent of the others without any overlap; otherwise parallelism suffers. For this, prior analysis of aliasing through indices is needed. In this connection, we give an overview of the solution for combining conditional constant propagation and interprocedural aliasing[10] that has to be modified so that it works for induction variables in addition to constants. Such a solution involves a fixpoint computation that computes both the set of induction variables and constants and the structure that corresponds to the SSA representation (see [6] for an example of a fixpoint iteration carried on a SSA structure): both these change as more accurate information is available. With such a solution, we can detect slices that are independent; give implicit or explicit name to each subslice and then assign each one of them a unique abstract location. This will then drive the fixpoint computation of computing aliases through slices with the previous subproblem fixpoint analyses as its inner core. When the fixpoint converges, we are done.

The condition on the subscripted variables is very stringent and may involve extensive analysis/computation. Partitioning an array so that as many non-overlapping slices as necessary for the needed accuracy is a difficult problem that has been addressed in the literature[3], [1], [11], [12]. To reduce the cost, we may follow any of the following strategies:

1: Use SSA representation for arrays with extensive induction variable analysis. But aliasing analysis needs to be done on the induction variables and other scalar variables beforehand. This requires an approach similar to Cytron[6] where constants are detected in the presence of aliasing. In our case, a fixpoint iteration on the SSA structure is needed for detecting induction variables in the presence of aliasing. This uncoupling of the aliasing analysis for scalars and structures will result in approximate but conservative information as the interactions between these has to be conservatively estimated. However, these interactions are also not very common.

2: Use memory disambiguation techniques (including omega test in conjunction with predicates to represent conditionals). 3: Use regular section descriptors[1] to subdivide an array into regular sections and provide abstract stack locations for them.

We now present alias analysis in the presence of pointers to array slices assuming that aliases are not transmitted between scalar and non-scalar variables. Due to lack of space, we do not present the solution to recognizing induction variables in the presence of aliases which is also needed.

3 Alias Analysis for Slices

Definitions and Preliminaries

Definite Alias: An abstract stack location x *definitely points-to* abstract stack location y if x and y each represent exactly the same set of real stack locations and the real stack location corresponding to x contains the starting address of the real stack location of y. It is represented as (x, y, D)

Possible Alias: An abstract stack x *possibly points-to* abstract stack location y if it is possible that one of the real stack locations corresponding to x contains the starting address of one of the real stack locations corresponding to y. It is represented as (x, y, P).

May Alias Pairs: To compute the may alias pairs at different program points, we construct a may alias graph (MAG) from the abstract stack. By applying transitive closure on this undirected graph, we get the may alias pairs. An alias relation (x, y) has an edge in MAG if any one of the following conditions holds true in the abstract stack:

x and y have a *definite* relationship.

x and y have a *possible* relationship.

x and z have a *possible* or *definite* relationship and z and y are aliased to slices or locations that overlap in the same array. Symbolic analysis (and more powerful methods[11]) is needed for accurate determination of overlap in the general case.

Must Alias Pairs To compute the must alias pairs at different program points, we construct a must alias graph (MUG) from the abstract stack and then apply transitive closure on this graph to get the must alias pairs. An alias relation (x, y) has an edge in MUG if x and y have a *definite* relationship in the abstract stack.

Dataflow Analysis

Intraprocedural analysis

In Fortran90, only explicit pointer assignment affect alias sets. The changes in the alias pairs can be represented using the following standard sets at each program point: GEN (the set of all aliases generated at the program point p), INPUT (The set of all aliases before p), KILL (the set of all aliases removed due to alias at p) and OUTPUT (the set of aliases after p). As an array pointer can be an another name for an array or array slice and other pointers can point to it, all pointers are normalized to indices on the base array by giving intermediate expressions names and changing bounds (possible as we use SSA).

Consider the assignment statement in Fortran90: $S : A \Rightarrow B$. Let A be a Fortran90 pointer and B be one of: a scalar, an array location of the form $X[c]$ with X a Fortran90 array, an array location of the from $X[i]$, an array slice of the form $X[i..j]$, a scalar pointer that points to a constant location, a pointer of the form $Y[e]$ with Y itself a Fortran90 pointer. Here i, j are variables, c a constant, e an expression. To capture all the various possibilities, we define *Llocs* (left locations) to be the set of abstract locations referred to by a reference on the lhs of an assignment and *Rlocs* (right locations) that of the rhs. We give the exact semantics for only some cases (due to lack of space) in Table 1; the cases omitted can be generated with some effort by looking at the ones listed. We have not explicitly given the semantics for the case where B is a pointer of the form $Y[e]$ with Y itself a Fortran90 pointer; we assume that all pointers are normalized to indices on the base array beforehand. Then

$KILL = \{(p, x, D) | (p, D) \in Llocs(A)\}$: Kill all relationships of definite $Llocs$.

$GEN = \{(p, x, d1 \wedge d2) | (p, d1) \in Llocs(A) \wedge (x, d2) \in Rlocs(B)\}$: Generate all possible relationships between $Llocs(A)$ and $Rlocs(B)$. Definite if both are definite.

$CHANGE = \{(p, x, D) | (p, P) \in Llocs(A) \wedge (p, x, D) \in INPUT\}$: Change from D to P, all relationships from possible $Llocs(A)$.

$OUTPUT = (INPUT - CHANGE) \cup \{(p, x, P) | (p, x, D) \in CHANGE\} - KILL \cup GEN$.

In conditional and loop statements, alias information flows from more than one point. At these points a conservative summary of the information flowing in has to be computed: computing the *points-to* alias information of the *then-body* and *else-body* independently and then merge the resulting alias sets in case of conditional and fixed-point iteration in case of loops. For select statement (similar to the C switch statement), we compute the alias information for each case of the select statement and then the output of all the case statements are merged.

If the variable is of the form $x.f1.f2 \ldots fn$, where x is a structure variable, $x.f1.f2 \ldots fn$ is given a unique location in the abstract stack and treated as any other variable. If x is a pointer to structure variable, then we retrieve $*x.f1.f2 \ldots fn$ from the abstract stack and compute aliases as for any other variable in the abstract stack.

Analysis of Gaussian Elimination

Assuming that induction variables and array indices will not be aliased (no cases of `j = *p + c`, etc.) in our implementation, we have successfully shown that the Gaussian elimination example given in Section 2 can be optimized as expected. The abstract stack is as follows:

S2: $< *x, A(i, i : n), D >, < *y, A(n, i : n), D >, < *z, A(i-1 : n, i-1), D >, < *temprow, heaploc, D >$

S8: $< *x, A(i, i : n), D >, < *y, A(j, i : n), D >, < *z, A(i : n, i), D >, < *temprow, heaploc, D >$

When the may and must aliases are computed from the abstract stack, we do not get spurious aliases between, say, x and y, as might be reported by less accurate aliases.

Interprocedural Analysis

The dataflow analysis consists of these steps [7]: constructing the invocation graph, followed by map, intraprocedural analysis and unmap for each function call. The C and Fortran90 versions do not differ in the interprocedural aspect as the Fortran90 specific aspects are present more in the intraprocedural part than in the interprocedural part (assuming the standard solutions for handling the aliasing through reference parameters in Fortran[5], [2]) We give a brief overview and omit details here except as relating to the ones we have attempted differently.

An invocation graph (IG) is constructed by performing a depth-first traversal of the call nodes of the abstract syntax tree (AST). Each call of a function has a unique path in the IG and hence we can distinguish between the different calls of the same function from different contexts. This approach leads to approximate analysis for recursive functions as at compile time we do not know the number of times a recursive function will be called.

In the construction of the IG we have 3 types of nodes: normal, approximate and recursive. We tag any non-recursive node as normal. When a recursive node has been identified the node is tagged as recursive node. Then a backward link is established to the ancestor node and it is tagged as approximate node.

In *map*, the alias information of the calling function is passed to the called function. As the called function affects the alias information of the callee in pointer based languages, the update of the alias set of the callee function is done through *unmap*. Our approach differs slightly in the case of invisible variables, i.e those variables whose alias relationships get modified although they may not be visible in the function.

Table 1: ***Llocs*** and ***Rlocs*** sets are relative to points-to set S; here $l..m$ = RANGE(i); $n..p$ = RANGE(j)

Variable Reference	L-location Set	R-location Set
& a	-	$\{(a, D)\}$
& $a.f$	-	$\{(a.f, D)\}$
& $a[c]$	-	$\{(a[c], D)\}$
& $a[i]$	-	$\{(a\{l..m\}, P)\}$
& $a[i..j]$	-	$\{(a\{l..m\}, P)\}$
a	$\{(a, D)\}$	$\{(x, d) \mid (a, x, d) \in S\}$
$a.f$	$\{(a.f, D)\}$	$\{(x, d) \mid (a.f, x, d) \in S\}$
$a[c]$	$\{(a[c], D)\}$	$\{(x, d) \mid (a[c], x, d) \in S\}$
$a[i]$	$\{(a\{l..m\}, P)\}$	$\{(x, P) \mid (a\{l..m\}, x, d) \in S\}$
$a[i..j]$	$\{(a\{l..p\}, P)\}$	-

4 Conclusions and Further Work

In this paper, we have presented an alias analysis for Fortran90 with the assumption that aliasing does not happen due to interactions between scalar and non-scalar variables. We have also developed solutions to two important subproblems that needs to be solved enroute: that of combining conditional constant propagation and induction variables with scalar interprocedural alias analysis. We are currently implementing the combined algorithm, both for detecting constants and induction variables. We also plan to study empirically the extent to which array slices and pointers are used in Fortran90 code and the usefulness of alias analysis in the presence of pointers to array slices. Additionally, range analysis optimization is also important and needs to be done in conjunction with alias analysis (see Gough et. al[13] who consider range analysis in the context of SSA).

References

[1] Vasanth Balasubramaniam. A technique for summarizing data access and its use in parallelism enhancing transformation. *SIGPLAN Notices*, June 1989.

[2] John Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proc of the Sixth ACM Symposium on POPL*, 1979.

[3] David Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. *SIGPLAN Notices*, 23(7):47–56, July 1988.

[4] Jong-Deok Choi, Micheal Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of poniter-induced aliases and side effects. In *Proc of the Twentieth ACM Symposium on POPL*, 1993.

[5] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Technical report, CS Dept TR87-61, Rice University, Oct 1987.

[6] Ron Cytron and Reid Gershbein. Efficient accomodation of may-alias information in SSA form. *SIGPLAN Notices*, 28(6):36–45, June 1993.

[7] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's Thesis, School of CS, McGill University, 1993.

[8] William A. Landi. Interprocedural aliasing in the presence of pointers. Technical report, PhD Thesis, Rutgers University, 1992.

[9] T.J. Marlowe, et.al. Pointer induced aliasing: a clarification. *ACM SIGPLAN Notices*, 28(8), 1993.

[10] K.S. Nandakumar. Combining conditional constant propagation and alias analysis. Master's Thesis, Indian Institute of Science, 1995.

[11] Bill Pugh. Omega test. *SIGPLAN Notices*, 27(7):140–151, July 1992.

[12] Michael Wolfe. Beyond induction variables. *SIGPLAN Notices*, 27(7):162–174, July 1992.

[13] Gough and Klaeren. Eliminating Range Checks Using SSA Form. CS Dept., QUT Australia,1995; http://www.dstc.qut.edu.au/ gough