Static interprocedural array shape analyses for refactoring and finding bugs in Fortran codebases

Mistral Contrastin

This document provides an exposition of Fortran arrays and potentital ways of improving reliability through static analysis. In particular it focuses on analysing Fortran's language features pertaining to arrays shapes and their misuse during procedure calls.

We first provide an introduction to Fortran arrays and their behaviour, then discuss potential problems enabled by the language features, and conclude by ideas about projects. If you have different ideas around the subject, please get in touch.

Background

Declaration

Fortran is a compiled language that require data to be declared before use. Here are three equivalent ways of declaring an array of length 10:

```
integer, dimension(10) :: arr ! Modern and preferred way
integer :: arr(10)            ! Older but still in use method
! Old method but still present in legacy code bases
integer arr
dimension arr(10)
```

For the rest of the discussion, we will use the first method with `dimension` *attribute*, but every example should work with the other methods.

Rank

Multi-dimensional arrays a first-level citizens and in Fortran parlour the number of dimensions of an array is called its *rank*. A 4 by 8 array of rank 2 is declared as follows

```
integer, dimension(4,8) :: arr
```

Although this is more directed towards performance debugging rather than reliability analysis, it is good to know that unlike other languages like C, Fortran arrays are column major.

Bounds

Unlike most languages Fortran arrays are indexed by 1 instead of 0 and more surprisingly this is easy (and common) to change within the language.

```
integer, dimension(10) :: arr ! Indices range 1 to 10 inclusive
integer, dimension(0:9) :: arr ! Indices range 0 to 9 inclusive
integer, dimension(-5:4) :: arr ! Indices range -5 to 4 inclusive
```

This can be independently to each dimension. The syntax `:` allows setting the lower and upper bound to the index range.

Shape

Shape of an array is a combination of its rank and its *extent*—size of each of its dimensions.

```
integer, dimension(-1:1,-2:2) :: arr ! arr has a shape of 3 5
integer, dimension(10) :: arr        ! arr has a shape of 10
```

Assumed shape and size

When Fortran arrays are declared in *subroutines* and *functions*, they need not be explicitly annotated with lower bound, upper bound, or size information. To use the array one has to rely on the other arguments passed to the subroutine or *intrinsic* functions (the relevant ones are listed a different section below).

```
subroutine Assumed(arr)
  integer, dimension(:) :: arr
  ! ...
end subroutine Assumed
```

The example above shows the modern way of doing it and the array is said to have an *assumed-shape*. The older (but still surviving in legacy code bases) version of this is *assumed-size* arrays which has a similar syntax except `*` is used in place of `:` to signify. Assumed-size arrays can only be used in the last dimension of a dimension attribute.

```
subroutine Assumed(arr)
  integer, dimension(10,20,*) :: arr
  ! ...
end subroutine Assumed
```

Restructuring arrays

Fortran allows shapes of arrays to be changed by one of two ways.

If the size of the array is known in advance, then the rank of the array can be changed in a procedure taking said array, so long as the sum of its extent before is greater than that of after reshaping. For example the following program works because 2 times 5 times 15 is less than 10 times 20 are equal.

```
program Reshape1
  integer, dimension(10,20) :: arr

  call Sub(arr)

  contains

  subroutine Sub(arr)
    integer, dimension(2,5,15) :: arr
  end subroutine Sub
end program Reshape1
```

In particular, an assumed-array declaration cannot be used to change the shape of an array, however, an assumed-size can be used. The following program compiles without warnings despite the fact that `Sub` has no way of knowing the procedure may require more space than what is passed inside.

```
subroutine Sub(arr)
  integer, dimension(2,5,*) :: arr
end subroutine Sub
```

It is, however, possible to restructure assumed-shape arrays after the declaration using the `RESHAPE` intrinsic. One can even use runtime values to do this as shown in the example below

```
subroutine Sub2(arr,n)
  integer :: n
  integer, dimension(:,:) :: arr
  integer, dimension(2,5,n) :: arr2

  arr2 = RESHAPE(arr, (/ 2, 5, n /))
end subroutine Sub2
```

Array initialisation

```
arr = (/ 1,2,3,4,5 /)       ! Through array literal
data arr/1,2,3,4,5/         ! Through data statement
! Through data statement but initialising cells explicitly
data arr(1)/1/, arr(2)/2/, arr(3)/3/, arr(4)/4/, arr(5)/5/
arr = 1                     ! Through vector assignment
arr(:) = 2                  ! Through slice
arr(1:5) = 3                ! Through slice with explicit bounds
arr = (/ (i*2, i=1, 5) /) ! With an inline loop
```

Array access

Array access is a vast subject in Fortran, so we will be introducing the bare minimum to aid our discussion.

The first thing to know is that Fortran is pass-by-reference meaning when one passes an array as an argument, the formal parameter will directly be accessing the passed array and not a copy of it (or a copy to a pointer to it *à la* Java).

Arrays can be accessed in various ways the most common way is `a(i)` where arr is an array type and `i` is an integer type. This looks like `b(i,j,k)` with the intuitive meaning when the array in multi-dimensional. Partially accessing a multi-dimensional array `b` with `b(1)` is not allowed, but `b(1,:,:)` produces the desired *slice*. Further, you can access some elements of an array. For example if the indices of `c` range from 1 to 10, `c(3:7)` evaluates to 5 element sized slice from the middle. One can even take all the even elements `c(1:10:2)` where 2 is the increment value. One can also use another array to retrieve elements of another array!

```
integer :: ix(5), arr(10), arr2(5)
! ...
arr = (/ (i*3, i=1, 10) /)
ix = (/ 1,2,3,9,10 /)
arr2(:) = arr(ix) ! arr2 now contains 3,6,9,27,30
```

Long story short, you can get very sophisticated (and obfuscating!) with Fortran array accesses very quick!

Intrinsics

Intrinsics refer to builtin functions as specified in by the Fortran standard. We explain a small fraction of the relevant array intrinsics here.

Name	Description
SHAPE	The shape of an array including its dimension sizes as an array
SIZE	The size of an array
LBOUND	The lowest possible index
UBOUND	The highest possible index
RESHAPE	Changes the shape of an array but keeping the contents

Intrinsics are useful in a static analysis context because they cannot be redefined in the language and might reveal useful information e.g. UBOUND for array upper bound will always be safe for the argument array in a loop, RESHAPE argument might be analysed to see if after reshape the indexing behaviour is erroneous.

Potential problems

Assumed-size size problem

Following program wouldn't cause a compile time error/warning and would terminate with a memory violation.

```
program AssumedSize
  integer, dimension(10) :: arr
  call sub(arr)

  contains

  subroutine sub(arr)
    integer, dimension(*) :: arr
    print *, arr(11)
  end subroutine sub
end program AssumedSize
```

Explicit bound and size passing

Below is an example of explicit bound passing and what might go wrong. Similar, mistakes occur with explicit size passing.

```
program ExplicitBound
  integer :: i
  ! its elements are numbers from 1 to 10
  integer, dimension(10) :: arr = (/ (i,i=1,10) /)
  ! the slice passed to the subroutine has elements from 1 to 9
  call sub(arr(1:9),LBOUND(arr, 1),UBOUND(arr, 1))

  contains

  subroutine sub(arr,l,u)
    integer :: l, u, i
    integer, dimension(:) :: arr

    do i = l, u
      print *, arr(i) ! prints 10 in the final iteration!
    end do
```

```
  end subroutine sub
end program ExplicitBound
```

If a static analysis simply propagates statically known values interprocedurally, this would be trivially detectable.

Reshaping error

This can go wrong in all sorts of ways, so please don't think the example below is definitive. In particular errors associated with this are not limited to statically memory allocated arrays.

```
program ReshapeError
  integer :: n, k
  integer, dimension(:), allocatable :: arr
  integer, dimension(:,:), allocatable :: arr2

  ! Allocate an array of size n and set its values to 1
  read *, n
  allocate(arr(n))
  arr = 1

  ! Let the second allocatable array be a partition of old one
  k = n / 2
  arr2 = RESHAPE(arr, (/ k, 2 /))

  ! If n == 0, this will produce garbage, this could have been detected
  ! statically
  print *, arr2(1,:)
end program ReshapeError
```

Avenues for projects

We have a Fortran corpus consisting scientific Fortran packages and well exceeding one million lines of code. It consists of open source as well as proprietary packages. Evaluation of any project is likely going to involve working on (a subset) of this corpus.

Verification (aka. bug finding)

Most problems outlined lead to an array being accessed out-of-bounds. With default compilation, this means the runtime will crush and burn and if the code is compiled with out-of-bounds checking flag (if that is even supported), the program will terminate with an exception.

Since (in old Fortran programs) there is a tradition of statically allocating data, it maybe posible to verify correctness of number of access patterns. In particular absence of errors in assumed-shaped arrays, reshaping, array initialisation can be shown. Alternatively, instead of trying to verify that these error are not present, one can also look for their violations, hence building a bug finding tool.

Such a project would be evaluated over corpus by either showing what percent arrays under various contexts are secure or finding out-of-bound access errors.

Refactoring tools

Some of the problems listed are due to interaction of old features of Fortran and can be prevented through refactoring strategies. This is important even for code that has old features and are correct as there is nothing preventing a novice programmer extending the code and introducing the error.

One example of this would be to convert all assumed-sized arrays to assumed-shaped ones. This can be done largely automatically. This would enable further refactorings such as elimination of explicit array sizes and bounds as arguments for the sole purpose of looping. This is error prone and as discussed below dangerous when bounds are remapped in declarations or only a slice of an array is passed.

For a refactoring project, the evaluation would involve finding sites that the problem refactoring is trying address and applying refactoring to it. A successful refactoring tool would be able to handle many of the sites and be able to explain why it is not safe to apply it for certain sites. This will, of course, be quantified. Because Fortran lacks formal semantics, it is difficult to do proofs of correctness, but a mathematical argument can be made by specifying assumptions about how constructors involved behave.