Supporting Arrays and Allocatables in LFortran

Gagandeep Singh

April 7, 2021

Contents

1 About Me

1.1 Contact Information

- **University** - Indian Institute of Technology, Jodhpur
- **Email IDs** - gdp.1807@gmail.com, singh.23@iitj.ac.in
- **Github** - czgdp1807
- **Gitlab** - czgdp18071
- **Timezone** - IST (UTC + 5:30)

1.2 Personal Background

I am a fourth year undergraduate in the department of Computer Science and Engineering at Indian Institute of Technology, Jodhpur.

I have completed the following relevant courses in my academic curriculum in the past years: Compiler Design, Computer Organisation and Architecture, Theory of Computation, Software Engineering, Object Oriented Analysis and Design.

1.3 Programming Background

I use Ubuntu 18.04.5 LTS as my operating system and Visual Studio Code as my editor. I have also worked with Eclipse IDE and Atom. I have written several lines of code in Python 3.x, C++11 and Java. I have also tried HTML, CSS and JavaScript in my initial years of undergraduate studies. A list of my ongoing and completed projects is as follows,

- **Enhancement of Statistics Module** - As a part of Google Summer of Code, 2019 with SymPy, in this project, I worked on designing an API framework for Stochastic Processes, Random Matrices and Multivariate Probability Distributions under the guidance of my mentor Francesco Bonazzi. I also developed and implemented recursive algorithms in Python 3.x for automatically solving probability and expectation queries for Markov Processes. The link to the report can be found here.

- **Anomaly Detection Engine** - This project was assigned to me during my internship at Morgan Stanley. I was responsible for designing and implementing an anomaly detection engine. Specifically, I implemented statistical algorithms in Java to detect anomalies in millions of data points consisting of strings, numerics and timestamps fetched from Elasticsearch servers using REST APIs. It is worthy to mention here that the anomaly detection engine is critical to teams involved in algorithmic trading to avoid economic losses due to anomalous data.

- **PyDataStructs** - As the name suggests it is a python package for data structures and algorithms. I started working on this in 2019 and since that point of time this project has successfully received contributions from various open source programs such as KWoC, GSSoC.

- **BNN** - BNN is a simple, light weight deep learning framework with a specialisation aimed for Binarized Neural Networks. I started working on this in 2020 and right now, I am heading towards implementing APIs for Layers, Activations and Models for MLPs.

Details on the above and more projects of mine are available on my Github profile. In all of the above projects I have used, `git` as my version control system. In addition, I have been using **Travis CI** in all of my projects except the one done at my internship.

1.4 Previous Contributions to LFortran

Following is the list of Merge Requests that I have made in the past to the Gitlab repository of LFortran,

1.4.1 Merged

- !735: Integer kinds
- !721: Adds pointers to LFortran
- !703: Added select case construct
- !698: Added Extraction of Kind from Integer Parameter
- !693: Support for BinOp and AnyTypeToReal for different Real Kinds
- !679: Implementing Real Kind
- !675: Logical types in LFortran
- !672: BinOp (Sub, Div, Mul, Pow) for Complex
- !666: Implemented Generic Code for ImplicitCast
- !671: Minor fixes for !666
- !657: Added Complex(expr, expr) to AST
- !771: Kinds for llvm::Type::getFloatTy and llvm::Type::getIntxxTy
- !722: Make integration_tests/arrays_01 work

1.4.2 Open

- !691: Draft: map->unordered_map in asr_to_llvm.cpp

2 The Project

In this section I would be discussing the project in detail. Specifically, I will be listing down features along with some examples, some of which are from the tests in LFortran repository. For each feature, I will also be providing some approaches and an analysis of advantages and disadvantages of one over the other.

I have divided this section into three parts, the first one covers arrays, the second one covers allocatables and the third one is covering some miscellaneous goals.

2.1 Arrays

2.1.1 Declaring Arrays

In 1 and 2, both single dimensional and multidimensional arrays of different types are declared. Note that these arrays are explicit shaped i.e., the details of each rank consisting of lower (defaults to 1) and upper bounds are available at compile time. Recently some progress has been made in the MR, !722 for adding declaration of arrays. The support for specifying dimensions for arrays is already available at AST and ASR level. Hence, the only challenging part left is to define a representation of arrays at LLVM level. After some discussions, we decided to use array descriptors because of the following reasons,

- They have already been tried in the past and are currently used by GFortran. Hence, they are reliable enough to be adapted by LFortran as well.
- It is possible to represent assumed shaped arrays as well which do not have details of each dimension available at compile time. For example, while passing arrays to functions/subroutines, we can just pass the descriptor which will be having all the details about the array and can be extracted inside the function as per requirements.
- It would be possible to check if the index in an array reference is within bounds or not during runtime. Hence, we would be able to alert the end users about such issues without throwing a segmentation fault which would help them in easily debugging their Fortran codes.

```
1 program array1
2 implicit none
3 real :: d, e(5), e2(2, 3), e3(1, 2, 3)
4 integer :: f, g(3), g2(6, 3), g3(4, 3, 2)
5 logical :: a, b(2), b2(3, 2), b3(2, 1, 2)
6 end program
```

Listing 1: tests/array1.f90

```
1 program array2
2 implicit none
3
4 real, dimension(5) :: a, b
```

```
5  integer, dimension(3) :: c
6  logical, dimension(2) :: d
7
8  real, dimension(2,3) :: e
9  integer, dimension(3,4) :: f
10 logical, dimension(5,2) :: g
11
12 real, dimension(2,3,4) :: h
13 integer, dimension(3,4,3) :: i
14 logical, dimension(5,2,2) :: j
15
16 end program
```

Listing 2: integration_tests/arrays_01.f90

As of now the descriptor of an array looks similar to the C++ structure shown in 3. Following are the points which summarise the meaning of each element in both the structures,

- `descriptor_dimension::lower_bound` - This specifies the lower bound of the dimension of an array only if the array is explicit shaped or the assumed shaped array has been allocated some space in heap memory. Otherwise it contains garbage value.
- `descriptor_dimension::upper_bound` - Similar to the above, this specifies the upper bound of the dimension of an array. It also contains garbage value if the shape of an array is not available at compile time or is not updated during run time via allocate or during a function call.
- `descriptor_dimension::dim_size` - It stores the size of a particular dimension which is computed using the upper and lower bounds. This element acts as a cache since, once computed at runtime, only one LLVM instruction will suffice to obtain the size of a dimension.
- `descriptor::array` - This contains a 1D `llvm::ArrayType`. If the array is explicit shaped then the number of elements are computed by multiplying the size of each dimension at compile time. Otherwise, the number of elements are kept as 0. For arrays whose memories are reserved using `malloc`, a simple element pointer can be used to refer to the base address by defining a new descriptor structure for such arrays.
- `descriptor::offset` - This contains the offset value. As of now this is always set to 0.

- `descriptor::dim` - This is simply the array of `descriptor_dimension` structure specifying the details of each dimension.

```
struct descriptor_dimension
{
  int64_t lower_bound;
  int64_t upper_bound;
  int64_t dim_size;
};

struct descriptor {
  ArrayType* array;
  int64_t offset;
  descriptor_dimension dim[Rank];
};
```

Listing 3: Array Descriptor for LFortran

For referring to an element inside an array, first the index values are combined using column major order (CMO) so that they are converted to appropriate index in 1D array and then the `descriptor::array` is referred using this new index. The reason for using CMO comes from the fact that Fortran uses CMO by default. However, we have planned to allow row major order (RMO) as an option as well.

As of now the above design has been implemented for integer type arrays. During the project more progress will be made on real and complex type arrays.

2.1.2 Operations on Arrays

Fortran allows operations on arrays in a way similar to the python library, numpy. An example is shown in 4. There are two approaches to add support for performing operations on arrays,

```
program array4
implicit none

integer :: a(3), b(3), c(3)

a = [1, 2, 3]

b = [1, 2, 3]

```

```
c = a + b
print *, c

c = a - b
print *, c

c = a*b
print *, c

c = a/b
print *, c

end program
```

Listing 4: Operations on Arrays

- **ASR Pass for array operations** - This approach involves performing an ASR pass to convert operations on arrays to a while loop over the indices of the operands. An example from one of Ondřej's comments is shown in 5.
- **Generating instructions at backend for array operations** - This approach involves checking if the operands in an expression have dimensions and then generating instructions for arrays as a special case.

The first method is simpler and is easy to implement without requiring any special treatment at the backend level. However, it leads to a larger ASR and avoids the scope of parallel computing by adding serialised loops by default. Though, an option can be provided for parallelizing loops and ASR pass in the first approach can be modified accordingly.

Overall, in my opinion, the first approach is good to be implemented for achieving a Minimum Viable Product (MVP) as early as possible. Moreover, it will also be helpful in debugging since an operation will be clearly visible as a loop when the ASR will be printed.

```
a = b + c

do j = 1, size(a,2)
do i = 1, size(a,1)
    a(i,j) = b(i,j) + c(i,j)
end do
end do
```

Listing 5: ASR Pass example for Array Operation

2.1.3 Indexing Arrays

I have discussed the details of how arrays are indexed in section 2.1.1. However, ideas for using those indexed values are yet to be explained. An example of the same is shown in 6 where indexed values of arrays are used in comparison operators, assignment statements.

```
program arrays_01
implicit none
integer :: i, a(3), b(4)
do i = 1, 3
    a(i) = i+10
end do
if (a(1) /= 11) error stop
if (a(2) /= 12) error stop
if (a(3) /= 13) error stop

do i = 11, 14
    b(i-10) = i
end do
if (b(1) /= 11) error stop
if (b(2) /= 12) error stop
if (b(3) /= 13) error stop
if (b(4) /= 14) error stop

do i = 1, 3
    b(i) = a(i)-10
end do
if (b(1) /= 1) error stop
if (b(2) /= 2) error stop
if (b(3) /= 3) error stop

b(4) = b(1)+b(2)+b(3)+a(1)
if (b(4) /= 17) error stop

b(4) = a(1)
if (b(4) /= 11) error stop
end
```

Listing 6: Example of Indexing Arrays in Fortran 90

Support for AST and ASR level is already available in the form of `ArrayRef` node. However at the backend level, when an array is indexed, the pointer associated with that index is obtained and not the value itself. Now, the assignment statement needs pointers but binary operators require the values

of array elements. Hence, it is necessary to decide when to load the value from the obtained pointer. As of now, when in an assignment statement, if an `ArrayRef` node is present on the right side then the value is loaded, otherwise not. For all other operators, load operation is always performed on the indexed pointer.

Another point to be noted is that, in MR !722, I haven't added checks yet to ensure that indices are within bounds. This will be done during the course of this project.

2.1.4 Passing Arrays as Function/Subroutine Arguments

There are a couple of tests, such as `integration_tests/arrays_03.f90`, and `integration_tests/arrays_04.f90`, where LFortran fails to generate instructions for passing arrays as function/subroutine arguments. I have shown a simple example since the bug is common among all the cases where the above feature is required. Specifically, the code in 7 leads to the assertion failure, `void llvm::CallInst::init(...): "Calling a function with bad signature!" failed`. In order to avoid this, in my opinion we need to define some protocol to allow arrays as arguments in functions/subroutines. One way to achieve this is to pass the runtime descriptor as an array to a function/subroutine. This will be exactly similar to what we currently do for complex types since essentially we are passing a structure type as an argument. Though, we will be required to take care of the following points,

```
1  program arrays_03
2  implicit none
3  integer :: x(10), y(10), i
4  do i = 1, size(x)
5      x(i) = i
6  end do
7  call copy_from_to(x, y)
8  print *, verify(x, y)
9
10 contains
11
12     subroutine copy_from_to(a, b)
13     integer, intent(in) :: a(:)
14     integer, intent(out) :: b(:)
15     integer :: i
16     do i = 1, size(a)
17         b(i) = a(i)
```

```
18      end do
19      end subroutine
20
21      logical function verify(a, b) result(r)
22      integer, intent(in) :: a(:), b(:)
23      integer :: i
24      r = .true.
25      do i = 1, size(a)
26          r = r .and. (a(i) .eq. b(i))
27      end do
28      end function
29
30  end
```

Listing 7: Passing arrays as function/subroutine arguments

- **Rank checking at compile time** - I have noticed that GFortran doesn't allow passing arrays to functions (which accept arrays) if the rank of the input array isn't matching with the parameter. This can be done at compile time while transitioning from AST to ASR level as the number of dimensions are available during this phase of compilation.

- **Pass descriptor as a pointer** - Though Fortran standard doesn't specify the details of the passing mechanism for arrays, instead of passing the descriptor structure of an array by value, we should pass it by reference. The reason is that this allows more efficiency. However, we will be required to ensure that arrays (with `intent(in)`) aren't modified during the duration of function execution as GFortran doesn't allow it. This can be done during the AST to ASR transition phase, since we will be required to check if the array is present on the left side of an assignment statement or not.

The above points apply to subroutine as well, with a slight difference that in this case, we would be compelled to pass the descriptor structures by reference in order to modify the values of any array with `intent(out)`.

2.1.5 Array Initializer Expressions

An example which shows different ways in which one can use array initializer expressions in Fortran is given in 8. Currently, LFortran raises the exception,

`LFortranException: visit_ImpliedDoLoop() not implemented` when transitioning from AST to ASR level. In the stacktrace, 9, it can be seen that `ArrayInitializer` node internally delegates the task to `ImpliedDoLoop` and since it is not implemented the exception is raised. Hence, in my opinion, we would be required to implement the said node at ASR and backend level to make things work. In addition, I believe that this node can also be used for operations on arrays. The other way would be to remove the `ImpliedDoLoop` node and just replace it with a single while or do loop in order to keep things standardised for both the features. Also, this would simplify the task of backend since we won't be required to pay special attention to array initializers.

```
program arrays_06
implicit none
real, dimension(6) :: x
real, dimension(10,10) :: A
integer :: i, j
x = [(i*2, i = 1, 6)]
print *, x
x = [(i+1, i*2, i = 1, 3)]
print *, x
x = [(i+1, i**2, i*2, i = 1, 2)]
print *, x
x = [(2*i, 3*i, 4*i, i+1, i**2, i*2, i = 2, 2)]
print *, x
A = 3
print '("Matrix A"/(10F8.2))', ((A(i,j), i = 1, 10), j = 1,
    10)
end program
```

Listing 8: Array Initializers

```
"/home/czgdp1807ssd/lfortran_project/lfortran/src/lfortran/
    ast.h", line 2058, in visit_expr_t<LFortran::BodyVisitor
    >()
     case exprType::ArrayInitializer: { v.
    visit_ArrayInitializer((const ArrayInitializer_t &)x);
    return; }
  File "/home/czgdp1807ssd/lfortran_project/lfortran/src/
    lfortran/semantics/ast_to_asr.cpp", line 1912, in LFortran
    ::BodyVisitor::visit_ArrayInitializer(LFortran::AST::
    ArrayInitializer_t const&)
     visit_expr(*x.m_args[i]);
  File "/home/czgdp1807ssd/lfortran_project/lfortran/src/
    lfortran/ast.h", line 2228, in LFortran::AST::BaseVisitor<
```

```
    LFortran::BodyVisitor>::visit_expr(LFortran::AST::expr_t
    const&)
6     void visit_expr(const expr_t &b) { visit_expr_t(b, self()
    ); }
7   File "/home/czgdp1807ssd/lfortran_project/lfortran/src/
    lfortran/ast.h", line 2059, in visit_expr_t<LFortran::
    BodyVisitor>()
8     case exprType::ImpliedDoLoop: { v.visit_ImpliedDoLoop((
    const ImpliedDoLoop_t &)x); return; }
9   File "/home/czgdp1807ssd/lfortran_project/lfortran/src/
    lfortran/ast.h", line 2238, in LFortran::AST::BaseVisitor<
    LFortran::BodyVisitor>::visit_ImpliedDoLoop(LFortran::AST
    ::ImpliedDoLoop_t const&)
10    void visit_ImpliedDoLoop(const ImpliedDoLoop_t & /* x */)
    { throw LFortran::LFortranException("visit_ImpliedDoLoop
    () not implemented"); }
11 LFortranException: visit_ImpliedDoLoop() not implemented
```

Listing 9: Array Initalizers Stacktrace from LFortran

2.1.6 Slicing Arrays

An example of slicing arrays is shown in 10. As of now there is no support for this feature at ASR level and hence, consequently at backend level in the master branch. This requires implementing `FuncCallOrArray` at ASR level. I would discuss more on the algorithms part for this feature instead of the design details because the latter is already fixed and will not have much impact on the feature itself.

For implementing slicing we should first note that no matter how many dimensions the array has, the result is always a 1D data structure and that too in CMO. So, if the expression is something like, `x(1:3:2, 1:3:2)` then, the result will be similar to the following order, `x(1, 1), x(3, 1), x(1, 3), x(3, 3)`. This means that to generate the list of sliced elements, first we fix the rightmost index and then recursively slice the first $n-1$ dimensions out of the total remaining n dimensions. In order to do this, one approach in my opinion is to convert the given slicing expression into a simple do loop in an ASR pass which implements the above logic. Another approach can be to implement a function in the C runtime library and use recursion to generate the sliced elements.

```
1 program arrays_07
2 implicit none
```

```
integer, dimension(3, 3, 3) :: x
integer :: i, j, z, k
z = 1
do i = 1, 3
    do j = 1, 3
        do k = 1, 3
            x(i, j, k) = z
            z = z + 1
        end do
    end do
end do
print *, x(1:3:2, 1:3:2, 1:3:2)
end program
```

Listing 10: Example for Array Slicing

2.1.7 Intrinsic Functions for Arrays

There are several intrinsic functions supported by Fortran which can be applied over arrays. Examples include, `sum`, `size`, `matmul`. Instead of discussing over the logic of how each function should be implemented, I would focus on the framework which should generalise well to any additional built-in functions we want to support for arrays,

- **Using Runtime Library** - We can use runtime library to implement functions over arrays in C by passing the descriptor structure to the C interface and implementing the logic there. We must ensure to successfully send the `llvm::ArrayType` to the runtime library.
- **Generating backend instructions using loops** - We can also treat intrinsic functions as something specialised and generate inline code for these using loops, etc. This can be done by replacing the occurrences of such built-in functions using an ASR pass and then the backend can process this new ASR without ever knowing about the existence of these functions.
- **Implementing in Fortran** - This idea comes from the discussions on supporting math library functions using `iso_c_binding`. Once that is done we can also implement all the intrinsic functions for arrays in Fortran itself.

For the long run the third idea is better, however for MVP, in my opinion we can use the first approach that is using C runtime library. We can also parallelize several functions using threads in C or by using any light weight dependency, though this is something we shouldn't care about much right now.

2.2 Allocatables

As of now, allocatables are parsed perfectly, however, ASR is not generated for them. An example is shown in 11 which leads to the exception `LFortranException: visit_Allocate() not implemented`. There are a couple of ways to implement this node. One approach is to just create a new type altogether for allocatables (such as `RealAllocatable`) and treat them separately by adding new code at ASR level instead of modifying the older one. An advantage of this method is that since we need to allocate memory on heap instead of stack, adding a new type would not disturb (and introduce bugs) the already existing framework which generates code for stack variables. The disadvantage is that the size of the LFortran's code base will increase a lot. Another method is to treat `allocatable` as an attribute in a way we treat other attributes. This will require adding attribute checks to not allocate memory on stack for these variables.

In the LLVM backend, we would be using `llvm::CallInst::CreateMalloc` for allocating memory on heap whenever, `allocate` is called on a symbol with `allocatable` attribute.

```
program allocate_01
implicit none
real, allocatable :: a(:), b(:,:), c(:,:,:)
integer :: n, ierr
n = 10
allocate(a(5))
allocate(b(n,n), c(n, 5, n), stat=ierr)
deallocate(a, c)
end
```

Listing 11: Example for Allocatables in Fortran

2.3 Miscellaneous Goals

Following are some examples which fail currently on the master branch of LFortran. I will try to fix these as well during the course of the project,

- `integration_tests/arrays_02.f90` - This test fails on master not due to a bug in arrays but due to the absence of support for `kind` function which seems like an inbuilt feature of Fortran 90 but is absent in LFortran. The exact error is `LFortranException: visit_FuncCallOrArray() not implemented`. I will work on implementing this method probably during the later stages of the project if the bug still persists until then.
- **Adding more support for Pointers** - Support for integer pointers has already been added, though for other types, we need to perform testing on corner cases and enhancing support iteratively.
- **Corner Cases for Kind** - Kind support is almost near to completion, it needs some more testing on corner cases and fixes if there are any unnoticed failures.

3 Timeline

In this section I will be presenting a tentative plan for adding the features discussed in the previous section. I will also be including pre-GSoC, and post-GSoC periods as a part of the timeline.

3.1 Pre-GSoC Period

This period includes all the time before the beginning of community bonding phase i.e., 17^{th} May, 2021. I will keep working on polishing my previous work on pointers, more support for kinds and fixing other bugs that come on the way. One such example is, #309.

3.2 GSoC Period

According to the official dates, the complete program can be divided into three parts namely, **Community Bonding Period**, **Phase - 1** and **Phase - 2**. The details of each of these is discussed in the following subsections.

3.2.1 Community Bonding Period

This part of the program starts from 17^{th} May, 2021 and ends at 7^{th} June, 2021 consisting of 3 weeks. The weekly plan for the duration is as follows,

- In the first week I will continue my work on adding support for declaring arrays. In my opinion a good approach to follow will be to first completely support one of the primitive types (such as `integer`) and then covering all other types following the same pattern.
- By the mid of second week, we can expect the work on declaring arrays to be almost complete. Hence, we would be able to start our work on array initialiser expressions. This will require some discussions before moving on towards the implementation of this idea. Since, it's just about choosing one of two approaches, I believe we will be able to start implementing array initialisers by the end of this week.
- In the third week, I will continue working on array initialisers. We will also start discussions on adding support for performing operations on arrays. Since this is an important feature, I will prefer to spend at least half of this week discussing various ways in which this feature can be implemented.

Overall, by the end of the community bonding period, work on declaring arrays would be complete, and array initialisers would be in the phase of implementation. Moreover, operations on arrays would be under discussions.

3.2.2 Phase 1

This phase starts from 7^{th} June, 2021 and ends at 16^{th} July, 2021, consisting of around 6 weeks. The weekly plan for the whole duration is as follows,

- By the mid of the first week, I will try to wrap up the discussions on operations on arrays. After that work on the same could be started. I will start off with implementing simple vector addition. In addition, the work on implementing array initialisers would be near to completion by the end of this week.
- In the second week and third week of this phase, I will continue working on completing operations after vector addition would be done. Since,

there are a bunch of cases to cover in operations, it will take some time in testing and formalising things.

- In the third week, I will also start my work on completing support for indexing arrays such as including runtime bound checks. This will require a good amount of testing to be performed to ensure that correct values are affected for given indices.

- In the fourth week, work on slicing arrays could be started. Since, this is more of an algorithmic and implementation focused task, not much discussions would be needed on this.

- By the beginning of fifth week, operations, indexing features for arrays would be near to completion and slicing implementation would be underway. Starting from mid of this week, we will discuss various approaches for allowing arrays to be passed as functions/subroutine arguments.

- During sixth week, which is basically evaluation period as well, we will focus on discussing passing arrays as function/subroutine arguments extensively so that we have solid implementation ideas for the next phase.

3.2.3 Phase 2

This would be the final phase of the program starting from 19^{th} July, 2021 and ending on 23^{rd} August, 2021 consisting of 5 weeks. The weekly plan for this duration is as follows,

- In the first week, we will be in a position to start implementing support for passing arrays as function/subroutine arguments. I will also start discussing finding a good way to implement a framework for intrinsic functions for arrays.

- During the second week, I will continue working on passing array as function arguments and would try to bring it near to completion. Moreover, by the mid of this week we will be able to start implementing support for intrinsic functions for arrays as well.

- During the third week, I will be working on intrinsic functions and discussions on allocatables would be underway.

- With the starting of the fourth week, I would start implementing allocatables. I will continue this till the end of the fifth week.

Overall, I have tried to provide a plan to fully accommodate the requirements of the program i.e., work equivalent to 20 hours per week. If needed I will try to shift some work from phase 1 to community bonding period to speed up the achievements of goals.

3.3 Post-GSoC Period

After GSoC, I will continue working on LFortran to make the release of MVP possible. After that, I will give time to adding optimisation features at ASR and backend level. Converting code from ASR to other languages such as C++, Python is also an interesting direction and I would like to work on it.

4 Acknowledgements

I would like to thank Ondřej for his continuous feedback on my Merge Request and on my approaches to tackle issues while working with LFortran.

5 References

- F2018 Interpretation Standard
- LLVM Language Reference Manual