Supporting Arrays and Allocatables in LFortran

International Fortran Conference, 2021

Gagandeep Singh, Software Developer (Quansight)
`gsingh@quansight.com`

Overview

- Background
- Array Declaration
- Operations involving arrays
- Allocatable arrays
- Array as input and output to functions/subroutines
- Automatic Deallocation

Background

Internal representations of code used by LFortran

- **Abstract Syntax Tree (AST)** - Contains all the syntax information in the input Fortran code. Each statement can be interpreted as a tree and then the whole code is just a forest of these trees.
- **Abstract Semantic Representation (ASR)** - Contains all the semantic information such as symbol tables (containing functions, variables, references to module elements, etc.). All the heavy lifting (type checking, implicit casting) is done here.
- **Backend** - Receives ASR as input and generates the code in desired language (LLVM, C++, etc.). My work involved dealing with LLVM backend.
- **ASR to ASR Passes** - Takes ASR as input and transforms it into an equivalent ASR. For example, converting all the loops to `while` loops, `select-case` to `if-else if-if` ladders which helps in simplifying backend.

Array Declaration

- All the dimensional and type information was already available in ASR representation of input code.
- We used a structure to represent arrays in LLVM IR. It contains the following information,
  - `ArrayType* array` - Pointer to 1D memory block containing the data.
  - `int64_t offset` - This contains the offset value. As of now this is always set to 0.
  - `dim` - This is simply the array of `dimension_descriptor` structure specifying the details of each dimension.

Array Declaration

- The `dimension_descriptor` structure has the following elements,
    - `lower_bound`
    - `upper_bound`
    - `size` - size of the current dimension
- For allocatable arrays, a 1 bit integer is also added to the array descriptor. It is used to check whether the pointer, `ArrayType* array`, is freed or not.

Operations involving Arrays

- **General approach** - Convert any array operation to loops. For example, `c = a + b` is converted to a loop, `c(i) = a(i) + b(i)`, for an iterator variable `i`.

- Achieved by writing ASR to ASR passes. Input ASR pass contains original expressions with operations on arrays and the output ASR contains the loops implementing those operations.

Allocatable arrays

- The descriptor for allocatable arrays is same as for “normal” arrays but contain an extra 1 bit integer to keep track whether the memory allocated is freed or not.
- We use `malloc` in `C` to allocate memory on heap. It is called in LLVM IR.
- Similarly, we use `free` in `C` to deallocate the memory allocated previously. In case of automatic deallocation (discussed later) we use the extra 1 bit integer to decide whether to call `free`.

Array as input and output to functions/subroutines

- Input to Functions/Subroutines - At LLVM level, pointer to the original array descriptor as passed as input to functions/subroutines.
- Output from Subroutines - Pointer to array descriptor is passed which can be modified by the subroutine.
- Output from Function - The function is first converted to a subroutine with the array being returned as `intent(out)` argument. Then any call to this function is converted to a subroutine call. Achieved by writing ASR to ASR pass. Helps in avoiding copying date from temporary return variable to the desired destination variable.

Automatic Deallocation

- Motivation
  - Free the memory on heap while leaving a scope (module, function, subroutine, program, etc.). Avoid double frees if already done explicitly by the user.
  - Before calling a function/subroutine, automatically deallocate arrays with `intent(out), allocatable` attributes.
- An `ImplicitDeallocate` node is appended to all the scopes in a ASR pass. It keeps track of only local variables. For example, input/output to a function/subroutine won't be affected.