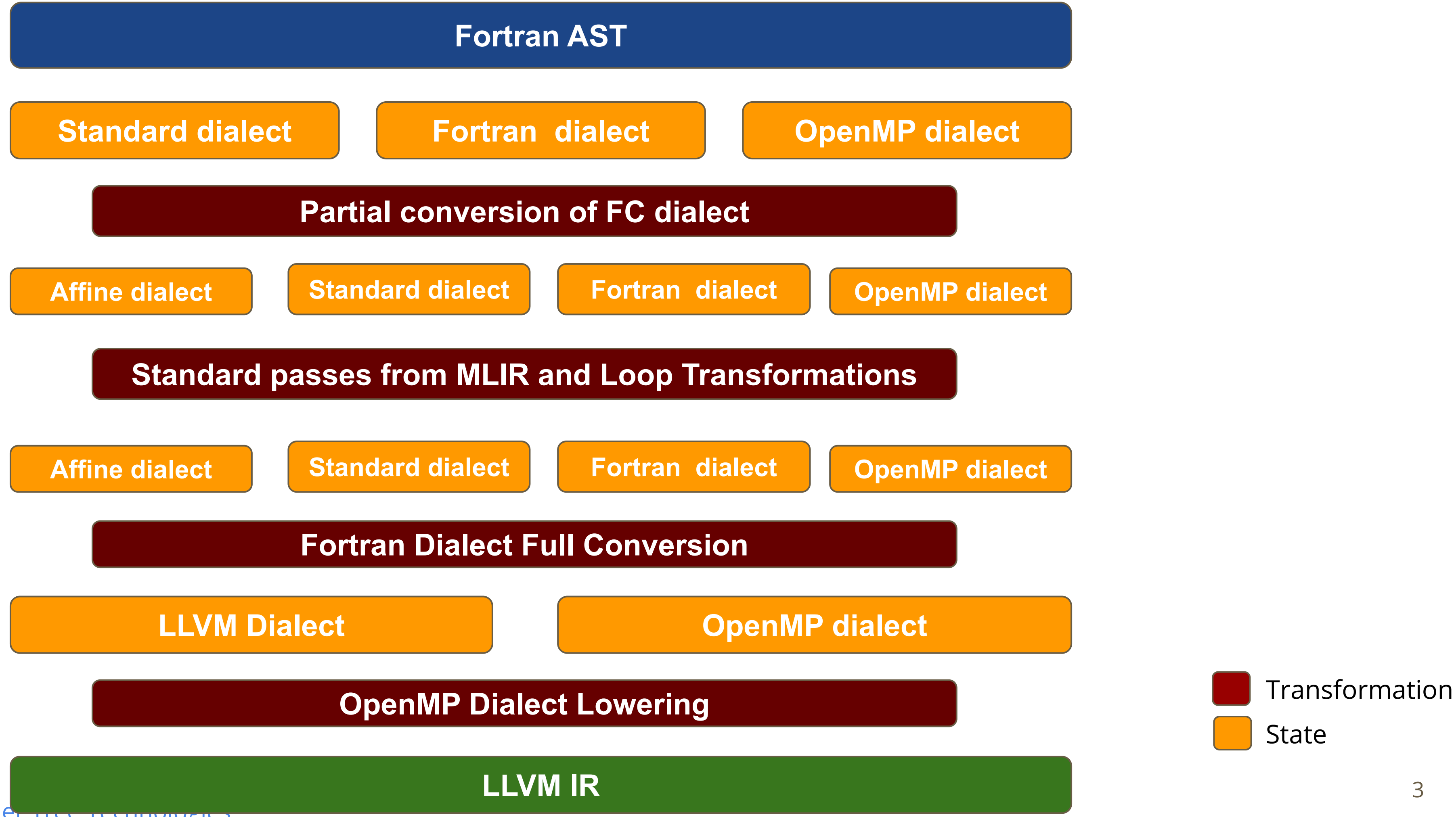
FC: MLIR Based Fortran Compiler

Compiler Tree Technologies.

Introduction

- Fortran frontend written in **C++11**, currently partially supports **f95**
- Built using the **MLIR** and **LLVM** infrastructures, with the same design principles as that of **Clang**
- Modular library based architecture. Easy integration with **Clang driver**
- Supports few **OpenMP clauses**
- Successfully compiles 400+ unit tests and **bwaves, exchange_r SPEC CPU 2017 benchmarks**


Fortran AST
Standard dialect
Fortran dialect
OpenMP dialect
Partial conversion of FC dialect
Affine dialect
Standard dialect
Fortran dialect
OpenMP dialect
Standard passes from MLIR and Loop Transformations
Affine dialect
Standard dialect
Fortran dialect
OpenMP dialect
Fortran Dialect Full Conversion
LLVM Dialect
OpenMP dialect
OpenMP Dialect Lowering
LLVM IR
Transformation
State

Compiler Stack

- Fortran frontend which partially supports Fortran 95+ standard
- Higher level MLIR
  - Fortran modules and function like operations
  - Fortran arrays
  - Fortran loop constructs
  - Custom loads / stores
  - Array Section
  - Standard dialect types / operations
- Lower level MLIR
  - Global variables
  - Affine / Loop dialect
  - Standard dialect types / operations
- OpenMP dialect to handle OpenMP related constructs
- LLVM Dialect / LLVM IR
- LLVM X86 CodeGen

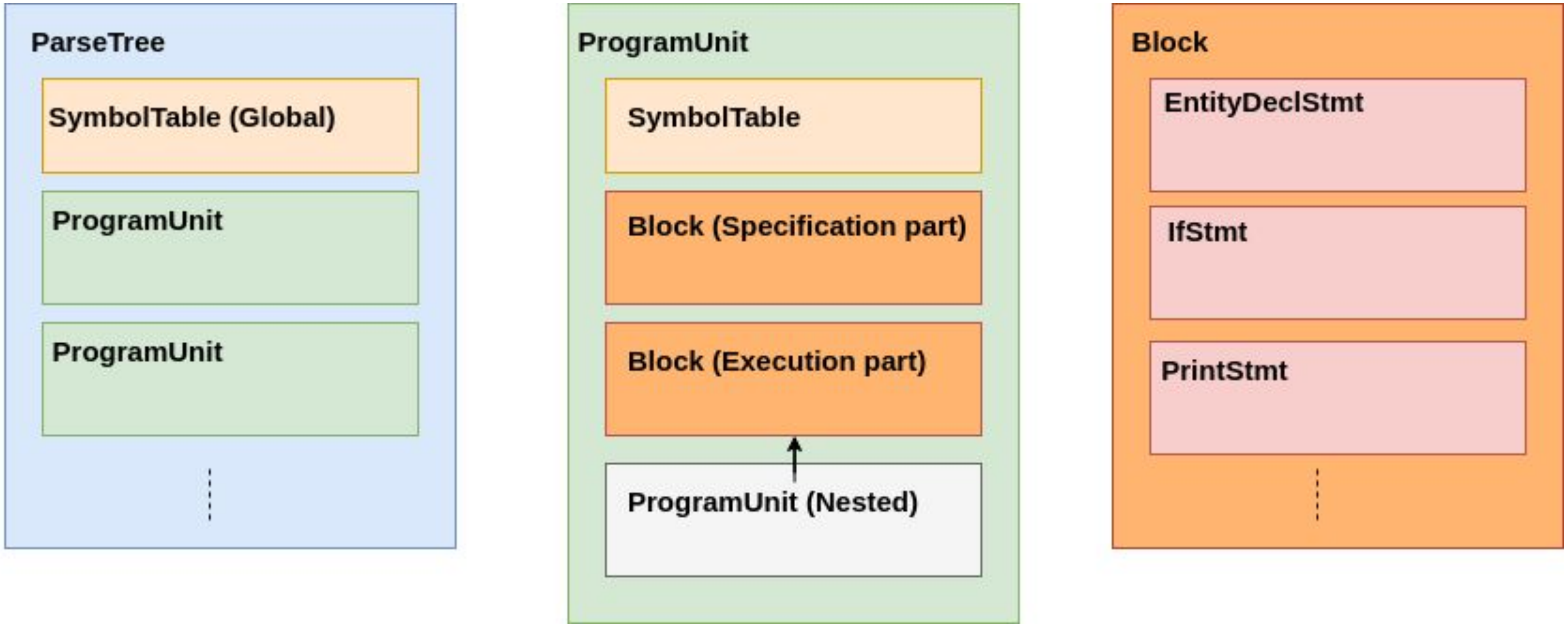
Frontend Design overview

- Built on LLVM Data Structures (ADT, Support, etc)
- Handwritten, recursive descent parser
- On-demand lexing
- Supports lookahead of N (usually, N<3) tokens
- Basic error reporting framework
- Syntax errors and very basic semantic errors are handled in Parser
- Memory management using LLVM Allocators
- Basic semantic checks performed after AST generation
- More semantic checks yet to be handled

Abstract Syntax Tree (AST)

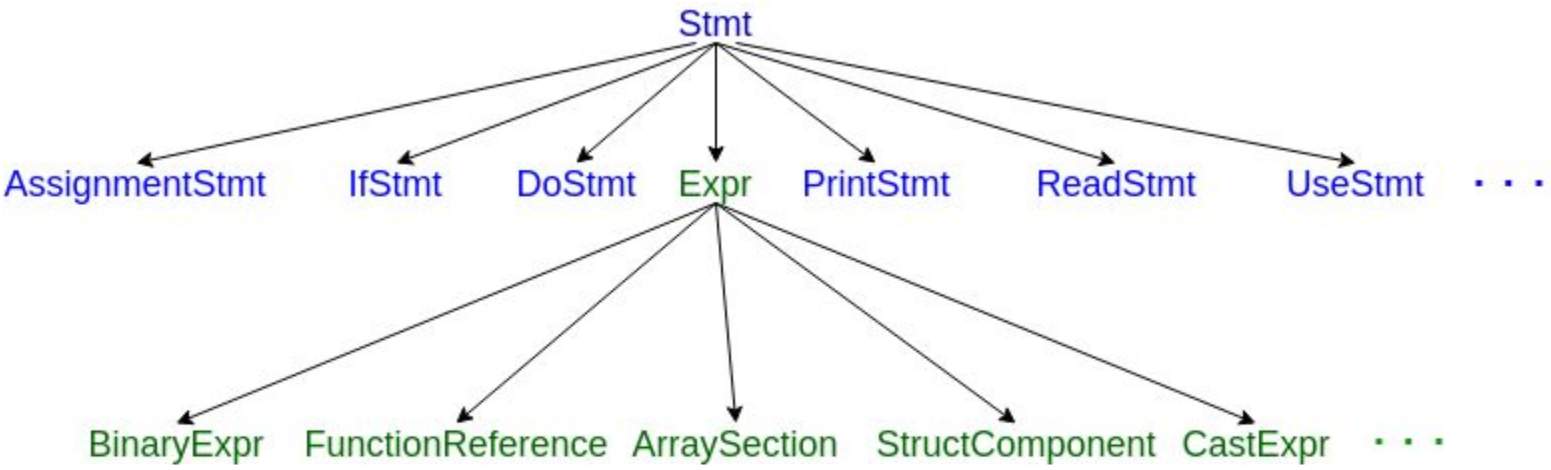
- Built using **LLVM ADT**: SmallVector, SmallPtrSet, StringRef, etc.
- Memory management of AST is done using **LLVMContext** like paradigm.
- A single compilation unit (fortran file) is held in **ParseTree**. (include/AST/ParseTree.h)
- ParseTree contains multiple **ProgramUnit**s. (ProgramUnit.h)
- A ProgramUnit can be **MainProgram**, **Function**, **Subroutine**, **Module**
- Each *ProgramUnit*s are made up of **Block**s,Nested *ProgramUnit*s and **SymbolTable**
- **Block** is a list of **Stmt**.
- Implemented under include/AST/ and lib/AST/

AST Structure: ProgramUnit and Block



AST Structure: Statements and Expressions

- **Stmt** is the basic entity for holding declaration and execution constructs.
- Expressions are held in **Expr,** which extends **Stmt**
- **Stmt** contains **operands** ,**source location** and a **pointer to parent Stmt**.
- Defined in include/AST/Stmt.h



AST Structure: Example

Example.f90

```
program example1
  integer(kind=4) :: a = 10
  print *, a
end
```

- ParseTree (Filename: Example1.f90)
  - ProgramUnit (MainProgram : example1, SourceLoc: (1, 9))
    - Block (SpecificationPart)
      - EntityDeclStmt (Symbol: "a", IntrinsicType: int32, SourceLoc: (2, 10))
        - Constant (10, SourceLoc: (2, 13))
    - Block (ExecutionPart)
      - PrintStmt (SourceLoc: (3, 4))
        - ObjectName (Symbol: "a", SourceLoc: (3,7))

AST: Types

- Closely resembles to that of MLIR.
- Maps Fortran types to MLIR types.
- Primitive types like Int32, Real, Logical and derived types includes types like ArrayType, FunctionType, StructType etc.
- Example Fortran types to fc::Type
    - integer (kind = 4) => fc::Type::Int32Ty
    - integer (kind = 8) => fc::Type::Int64Ty
    - real => fc::Type::RealTy

SymbolTable

- Holds name vs. **Symbol** mapping using std::map in the given scope.
- Current Scope kinds are **GlobalScope**, **FunctionScope**, **ModuleScope**, **MainProgramScope**. (DerivedTypeScope?)
- Contains reference to the ProgramUnit and vice-versa.
- Contains **pointer to the parent table**. Example: MainProgramScope table has GlobalScope as parent table.
- SymbolTables can be **serialized/ de-serialized** (Helpful for extern module references)
- Implemeneted in include/AST/SymbolTable.h

Symbol

- Entities such as variable names, Function names, Module names.
- Attributes
    - Name
    - Type
    - SymbolTable it is residing in
    - SourceLocation of the Symbol Declaration/defintion
    - ParentSymbol
    - Other Fortran related attributes like, , intent kind, pointer type/ target type, linkage type, constant, allocation type (global/local)
- A Symbol can have a parent Symbol when the original declaration happens in a different Scope. Example: ModuleScope variable used in MainProgramScope
- Implemented in include/AST/SymbolTable.h

Readable AST Dump for Example.f90

```
Program: Example.f90
// GlobalScope, Parent: None
SymbolTable Global {
  // Symbol List:
  // (1, 9)
  // ID: 1, NonConstant, NonAllocatable, NonTarget, NonPointer, Alloc_None, Intent_None, Global
  (int32)() example1
}
// MainProgram
int32 example1() {
  // MainProgramScope, Parent: GlobalScope
  SymbolTable example1 {
    // Symbol List:
    // ID: 2, NonConstant, NonAllocatable, NonTarget, NonPointer, StaticLocal, Intent_None, example1
    int32 a
  }
  // Specification Constructs:
  EntityDeclList {
    {
      // (2, 14)
      NAME: a
      SYMBOL ID: 2
      INIT: 10
    }
  }
  // Execution Constructs:
  // (3, 3)
  printf(a)
}
```

Implemented in ParseTreeDumper.cpp

AST Traversal: Passes!

- LLVM Pass like structure has been added to traverse the higher level AST structures like Program Units and Blocks
  - **ASTProgramPass**: runOnProgram(ParseTree* tree)
  - **ASTPUPass**: runOnProgramUnit(): Runs on all the ProgramUnits in the current AST.
  - **ASTModuePass**: runonModule(): Runs on all the **Module**s in the AST
  - **ASTBlockPass: runonBlock():** Runs on all the **Block**s in all the ProgramUnit.
- **ASTPassManager** is a simple class to collect and run all the passes.
- Sema, AST Expansion and Codegen phases are all AST passes!
- Implemented in include/AST/ASTPass.h

AST Traversal: StmtVisitor

- Visitor patterns are provided to traverse the Stmt (hence, also Expression) in DFS order.
- Used with the AST Passes on Block level.
- **Any node in the AST can be modified using the visitor.**
- Implemented in include/AST/StmtVisitor.h

Modifying the AST

- **Adding a Stmt**: **ParseTreeBuilder**
  - Similar to LLVM IRBuilder.
  - Only way to build the AST.
  - Handles the memory. Uses LLVM Allocator.
  - New Nodes should be inserted in the **parent Block** of the current **Stmt**
- **Deleting a Stmt:**
  - No need of explicit de-allocation
  - Replacing the Stmt reference in parent Stmt/Block will work.
- **Updating a Stmt:**
  - **Stmt** kind cannot be updated. It should be replaced.
  - Updating the operands of the current Stmt with the new one.
- In the similar way, Program Units can be added/deleted/ udpated.

Semantic analysis - Sema (lib/sema)

- Sema in FC is implemented as set of passes, where each pass will perform some specific transformation on AST.
- Each pass performs only specific task, for example resolving type of an expression.
- Pass infrastructure is implemented using visitor classes, where each expression, statement or construct is recursively visited.
- Design allows pass to create new nodes, replace nodes and modify AST nodes.
- Passes are used for resolving type of symbols, expanding constructs, expanding intrinsics, handling modules etc.

Module Handling (ModFileDumperPass.cpp)

- .mod file are like the “header” files of C in Fortran.
- Tracks variables, derived types etc.
- “Finer” global scope (ie. only PUs who use it)
- Persistent data.
- Parsed like any other PU, but get’s dumped at semantics.

Example:

```
module mod
  ! variable decl, DTDs etc.
end module mod
```

Gets dumped by semantics to a text file with the name: <module-name>.mod

Read to get the symtab and DTDs imported to a PU, eg:

```
Program foo
  Use mod
End program foo
```

Symbol Resolution & Type Verifier (SymbolResolverPass.cpp)

- Multiple passes are used to resolve symbols, their scope, type and expression type.
- When parser can not decide the base type of an expression, such expressions are parsed into one common type and later are resolved in sema. For example parser can not differentiate between function reference and array element.
- Type of symbols with module scope are also not known to parser.
- In all those cases where parser can not decide type, expression type and scope of any symbol/expression, a pass in sema is added.
- Cast expressions are generated to match the type of LHS.

Symbol Resolution & Type Verifier (2)

- In below example, during parsing we don't know the type of mass.
- Type of symbol **mass** is resolved during sema pass.
- After type is resolved cast expressions are generated for RHS to match type of LHS

```
...
use globals
  integer :: vol = 10
  mass = vol * 10
...
```

```
// (3, 14)
vol = 10
// (4, 3)
t.2 = vol * 10
t.1 = cast t.2 to   real
mass = t.1
```

Constant Propagation (ConstPropPass.cpp)

- Evaluates and substitutes arithmetic and logical expressions involving literal constants and param variables.

```
real, parameter :: c1 = 142856.0 + 1.0
real, parameter :: c2 = 10.0 * 1000.0
real, parameter :: x = c1 / (c2 * 100)

if (.true. .and. (1 > 17)) then
  print *, (72 + 341 * 496) / 6508
end if
```

```
// (2, 22)
c1 = 142857.000000
// (3, 22)
c2 = 10000.000000
// (4, 22)
x = 0.142857
// (6, 3)
if (.false.) {
  // (7, 5)
  printf(26)
}
```

AST Expansion: Constructs (ConstructExpander.cpp)

- Some AST nodes, which can not be directly represented in LLVM - IR are expanded.
- Array sections, where-constructs, for-all and select-case constructs (AST nodes) are expanded in sema passes.
- We are planning to move this to MLIR

```
...
where ( a > 3 )
     a = 3
elsewhere
     a = 10
end where
...
```

⇨

```
do (i..) {
     if (a(i) > 3) {
       a(i) = 3
     }
     else {
       a(i) = 10
     }
}
```

AST Expansion: ArraySections (ArraySectionExpander.cpp)

```
program test
  Integer :: array(10, 10)
  integer, parameter :: l =
1
  integer, parameter :: u =
10
  array(:, :) = 10
end program test
```

```
...
  l = 1
  // (4, 25)
  u = 10
  // (5, 3)
  t.1 = (/*IndVar=*/test.tmp.1, /*Init=*/1, /*End=*/10,
/*Incr=*/1)
  do (t.1) {
    // (5, 3)
    t.2 = (/*IndVar=*/test.tmp.0, /*Init=*/1,/*End=*/10,
/*Incr=*/1)
    do (t.2) {
      // (5, 3)
      array(test.tmp.0, test.tmp.1) = 10
    }
  }
...
```

AST Expansion: for-all

```
program test
  integer ::i,a(10, 10), j
  a = 0
  forall(i=1:10, j=1:10, a(i, j) == 0) a(i, j) = 1
  print *, a
end program test
```

```
...
t.3 = (/*IndVar=*/i, /*Init=*/1, /*End=*/10,
/*Incr=*/1)
  do (t.3) {
    // (4, 3)
    t.4 = (/*IndVar=*/j, /*Init=*/1, /*End=*/10,
/*Incr=*/1)
    do (t.4) {
      // (4, 3)
      t.5 = a(i, j) == 0
      if (t.5) {
        // (4, 40)
        a(i, j) = 1
      }
    }
  }
...
```

AST Expansion: Intrinsics (IntrinsicExpanderPass.cpp)

- FC supports some of the intrinsics from FORTRAN 90.
- Support for these intrinsics are added by expanding/reducing the intrinsic into equivalent AST nodes.
- We currently support around 39 intrinsics, covering count, huge, maxloc, minloc etc.
- Math intrinsics with corresponding llvm intrinsics are not expanded, instead are handled in codegen.
- Array bound intrinsics for static arrays are reduced in sema.

AST Expansion: Intrinsics example

```
program pgm
  integer::
a=1,b=10,c=-3,d=4
  print *, min(a,b,d)
end program pgm
```

```
 ...
  vin.tmp.0 = a
  // (3, 12)
  t.1 = b < vin.tmp.0
  if (t.1) {
    // (3, 12)
    vin.tmp.0 = b
  }
  // (3, 12)
  t.2 = d < vin.tmp.0
  if (t.2) {
    // (3, 12)
    vin.tmp.0 = d
  }
  // (3, 3)
  printf(vin.tmp.0)
...
```

AST Expansion: IO and Format

WriteStmts with formats are expanded in sema. We try to resolve formats during compile time itself. For example,

```
write(*, '(9i1)')
         ((array(i,j),  j = 1, 9), i = 1, 9)
```

```
do i=1, 9
  do j = 1, 9
    print(array(i, j)
  end do
  print("\n")
end do
print("\n")
```

CodeGen (lib/codegen)

- Generates the High level MLIR from AST
- AST Type to MLIR type systems conversion is trivial for basic, derived types.
- External Function, Subroutine and MainProgram ProgramUnits are converted to FCFuncOp
- Fortran Modules are converted FortranModuleOp.
- CGExpr.cpp contains expression handling.
- CGStmt.cpp contains Fortran executable statements handling
- CGASTHelper.cpp contains AST related helper routines for MLIR generation

High level MLIR Operations

Fortran dialect: Types

- Mostly from Standard dialect
- **RefType** : Reference to region of memory
  - Similar to memref in standard dialect
- **ArrayType**: Represents fortran arrays, contiguous memory with column major layout

integer :: array(1:10, -3:30) → fc.array<1:10 x -3:30 x i32>

real(kind=8):: array2(:,:) → fc.array<? x ? x f64>

Fortran Dialect: Memory related operations

- Works on RefType
- Alloca and dealloca operations
  - Allocate either on stack or on heap based on an attribute
  - Contains attributes like “**static**”, “**captured”**, etc.
  - Dealloc operations are managed accordingly.
- Load and Store operations
  - Can be used to Load / Store **scalar or array element or array section**
  - Uses AffineMap to hold subscript expression.

Fortran Dialect: Function like operations

- Holds Fortran program, subroutine and function
- `FunctionLike OpTrait`
- More information than `mlir::FuncOp`
    - Nested functions
    - Used / Captured modules
    - Captured variables from parent
- Nested functions are placed in entry block
    - Different from closures: Doesn't capture parent function variables.

Fortran Dialect: Function like operations

```fortran
subroutine add(a, b)
  integer :: a, b
  a = a  + b
end subroutine add
```

```
fc.function @add(%arg0: !fc.ref<i32>, %arg1: !fc.ref<i32>) {
    %0 = fc.load %arg0 {name = "a"} : i32
    %1 = fc.load %arg1 {name = "b"} : i32
    %2 = addi %0, %1 : i32
    fc.store %2, %arg0 {name = "a"} : !fc.ref<i32>
    fc.return
}
```

Fortran Dialect: Nested functions

```
subroutine sub1
  integer :: a
  print *, func()
contains
  integer function func
    func = a
  end function func
end
```

```
fc.function @sub1() {
  fc.function @func() -> i32 {
    %2 = fc.get_element_ref @a::@sub1 : !fc.ref<i32>
    %3 = fc.allocate func : !fc.ref<i32>
    %4 = fc.load %2 {name = "a"} : i32
    fc.store %4, %3 {name = "func"} : !fc.ref<i32>
    %5 = fc.load %3 : i32
    fc.return %5
  }
  %0 = fc.allocate a, implicitly_captured : !fc.ref<i32>
  %1 = fc.call @func::@sub1() : i32,
  fc.print %1 {arg_info = #fc.is_string< >}
  fc.return
}
```

Fortran Dialect: get_element_ref and call

- Used to access the variables/functions from different Scope
  - Parent function variables
  - Module variables.
- Uses `mlir::SymbolRefAttr`
- Retains high level information
- Helps in easier Analysis
  - Example: Alias Analysis (discussed later)

Fortran Dialect: Fortran Modules

```
module mod1
  integer :: a
contains
  integer function func
    func = a
  end function func
end module mod1
```

```
fc.fortran_module mod1 {
  %0 = fc.allocate a, static : !fc.ref<i32>
  fc.function @func() -> i32 {
    %1 = fc.get_element_ref @a::@mod1 : !fc.ref<i32>
    %2 = fc.allocate func : !fc.ref<i32>
    %3 = fc.load %1 {name = "a"} : i32
    fc.store %3, %2 {name = "func"} : !fc.ref<i32>
    %4 = fc.load %2 : i32
    fc.return %4
  }
}
```

Fortran Dialect: Loops

```
do i = 1, 10
   array(i) = i
enddo
```

```
fc.do %arg0 = %3, %c10_i32_0, %c1_i32_1  {
   %4 = index_cast %arg0 : index to i32
   fc.store %4, %1[%arg0] : !fc.ref<!fc.array<1:10 x i32>>
   ...
 } enddo
```

Fortran Dialect: Loops : Handling control flow

```
do i = 1, 10
  if (i > 3) a = a + 2
enddo
```

```
fc.do %arg0 = %3, %c10_i32_0, %c1_i32_1  {
   %c3_i32 = constant 3 : i32
   %4 = "fc.cast"(%arg0) : (index) -> i32
   %5 = cmpi "sgt", %4, %c3_i32 : i32
   loop.if %5 {
     %c2_i32 = constant 2 : i32
     fc.store %c2_i32, %0 {name = "a"} : !fc.ref<i32>
   }
   ...
 } enddo
```

Fortran Dialect: Irregular Loops

- AST is converted to CFG based loops
- Evaluating affine.graybox like operation
- AST pass to convert them to standard loop format
- Example:

```
outer: do
  do i = 1, 10
    if (i > 6) exit outer
    if (i > 3) array(i) = 3 + a
    ...
  enddo
enddo outer
```

- Note that fc.do operation doesn't work here because **exit outer** needs to transfer control to the **enddo outer** statement. Which means **i** will be not updated.

Fortran Dialect: Array Section Handling

- Fortran Array: Contiguous memory with a lower bound index and an upper bound index
- First class / SSA representation of Fortran array sections
- An array operation:
  - ```
    integer :: a(10), b(10)
    ; initialize a …
    b = a + a
    ```
  - ```
    %7 = fc.load %4[] {name = "a"} : !fc.array<1:10 x i32>
    %8 = fc.load %4[] {name = "a"} : !fc.array<1:10 x i32>
    %9 = fc.array_addi  %7, %8 : !fc.array<1:10 x i32>
    fc.store %9, %3[] : !fc.ref<!fc.array<1:10 x i32>>
    ```
- Scalar optimizations such as CSE can be done on array operations
- Similar to **Tensor**, but with lower and upper bound indices

Fortran Dialect: IO operations

- Fortran IO statements like read, write, print, etc.

```
print *, array
read *, a
write(6, *)  a
```

```
fc.print %3 {arg_info = #fc.is_string< >}
%4 = fc.read %c5_i32, %2 {arg_info = #fc.is_string< >}
%5 = fc.load %2 {name = "a"} : i32
fc.write %c6_i32, %5 {arg_info = #fc.is_string< >,
                                      space_list = [-1 : i32]}
```

OpenMP Dialect : ParallelOp

```
a = 10
b = 20
!$omp parallel
print *, "hellofrom", a, b
!$omp end parallel
```

```
omp.parallel(%2, %3)  {
        %5 = fc.constant_string("hellofrom"):
                         !fc.array<0:9 x i8>
        %6 = fc.load %2 {name = "a"} : i32
        %7 = fc.load %3 {name = "b"} : i32
        fc.print %5, %6, %7 {arg_info = #fc.is_string< 0 >}
}
```

High level MLIR Transformations

Basic optimizations

- **Memory to Register**
  - Promotes suitable `fc.allocate` to register
  - Prototype implementation, doesn't work for nested regions yet.
- **Simplify CFG**
  - Remove dead blocks, etc
- **LICM**
  - Hoisting operations out of Loop like interface (fc.do)
  - Includes memory operations
  - Based on Alias Analysis results.
- Other simple peephole transformations to enable Loop Nest Optimizations

Alias Analysis Framework (AliasAnalysis.cpp)

- Prototype similar to llvm **basic-aa**
- Basic **AliasSetTracker** class for caching / tracking alias results
- Simpler analysis because of higher level structure / operations closer to the language specification
- Users
  - Hoisting / Sinking transformations on load / store operations
  - Dependence Analysis
  - Other memory operation related transformations

Lowering to Affine / Standard Dialect

Lowering: fc.function (LowerProgramUnit.cpp)

- Conversion to `mlir::FuncOp`
- Flattens the nested functions with appropriate name mangling

```
fc.function @pgm() -> i32 {
  fc.function @sub1() {
    %1 = fc.get_element_ref @a::@pgm : fc.ref<i32>
    %2 = fc.load %1 {name = "a"} : i32
    fc.print %2 {arg_info = #fc.is_string< >}
    fc.return
  }
  %0 = fc.allocate a, implicitly_captured :
                      !fc.ref<i32>
  %c0_i32 = constant 0 : i32
  fc.call @sub1::@pgm()
  fc.return %c0_i32
}
```

```
func @pgm() -> i32 {
  %c0_i32 = constant 0 : i32
  %2 = fc.allocate a : !fc.ref<i32>
   . . .
  call @pgm.sub1(%2)
  return %c0_i32 : i32
}

func @pgm.sub1(%arg0: !fc.ref<i32>) {
  %2 = fc.load %arg0 {name = "a"} : i32
  fc.print %2 {arg_info = #fc.is_string< >}
  return
}
```

FC dialect: Global variables

- Module variables and variables with **save** attribute are implemented as globals.
- No native support in MLIR for global variables
- GlobalOp
  - SImilar to `LLVM::GlobalOp` implementation
  - Contains initializer region
  - Placed under `mlir::ModuleOp`
- AddressOfOp
  - Similar to `LLVM::AddressOfOp`
  - Contains `mlir::SymRefAttr` operand which points to `FC::GlobalOp`
  - Used in functions to access global variables.

Lowering: fc.fortran_module

- Module variables become global variables
- Flattens nested functions with appropriate mangling

```
fc.fortran_module mod1 {
  %0 = fc.allocate a, static {10 : i32}
            : !fc.ref<i32>
  fc.function @sub1() {
    %1 = fc.get_element_ref @a::@mod1
            : !fc.ref<i32>
    %2 = fc.load %1 {name = "a"} : i32
    fc.print %2 {arg_info = #fc.is_string< >}
    fc.return
  }
}
```

```
%0 = fc.global mod1.a { 10 : i32 } : !fc.ref<i32>
func @mod1.sub1() {
  %1 = fc.addressOf @mod1.a : !fc.ref<i32>
  %2 = fc.load %1 {name = "a"} : i32
  fc.print %2 {arg_info = #fc.is_string< >}
  return
}
```

Lowering: Array Section operations (ArrayOpsLowering.cpp)

- Array operations are expanded into `fc.do` loops with scalar operations
  - Array Op:

    ```
    %9 = fc.array_addi  %7, %8 : !fc.array<1:10 x i32>
    ```
  - Array Op lowered to fc.do:

    ```
    fc.do %arg0 = %7, %8, %c1_i32 {construct_name = "arrayop"}  {
      %9 = fc.load %3[%arg0] {range_info = #fc.subscript_range< 0 >} : i32
      %10 = fc.load %2[%arg0] {range_info = #fc.subscript_range< 0 >} : i32
      %11 = addi %9, %10 : i32
      fc.store %11, %3[%arg0] {range_info = #fc.subscript_range< 0 >}
    } enddo {construct_name = "arrayop"}
    ```
- Memory is allocated for SSA (array) values if not already allocated
- Opportunity to combine multiple ops into one loop to effect loop fusion

Lowering : fc.do to affine.for

- `fc.do` is lowered to AffineForOp to enable loop transformations
- Load, store and arithmetic operations are converted to corresponding affine operations

```
fc.do %arg15 = %c1_i32, %5, %c1_i32  {
  fc.do %arg16 = %c1_i32, %9, %c1_i32  {
     fc.store %cst, %arg0[...]
  } enddo
} enddo
```

```
affine.for %arg15 = 1 to ()[s0] -> (s0 + 1)()[%8] {
 affine.for %arg16 = 1 to ()[s0] -> (s0 + 1)()[%13] {
   affine.store %cst, %24[...] : ...
  }
}
```

Lowering: FC memory operations to affine dialect

- Converts `fc.RefType` to `std.memref` using `fc.cast_to_memref` operation
- Preparation for LNO

```
%8 = subi %6, %7 : i32
%9 = index_cast %8 : i32 to index
%10 = fc.load %2[%9] {name = "a",...
….

%13 = addi %11, %12 : i32
%14 = index_cast %13 : i32 to index
fc.store %c12_i32, %2[%14] {name = "a",
```

```
#map1 = ()[s0, s1] -> (s0 - s1)
#map2 = ()[s0, s1] -> (s0 + s1)
%8 = index_cast %6 : i32 to index
%9 = index_cast %7 : i32 to index
%10 = fc.cast_to_memref %2 : memref<10xi32, #map0>
%11 = affine.load %10[symbol(%8) - symbol(%9)]
…
%14 = index_cast %12 : i32 to index
%15 = index_cast %13 : i32 to index
%16 = fc.cast_to_memref %2 : memref<10xi32, #map0>
affine.store %c12_i32, %16[symbol(%14) + symbol(%15)]
```

Lowering to LLVM Dialect

LLVM Lowering (lib/FCToLLVM/)

- Affine and Standard dialects are lowered using standard conversion patterns
- Full Conversion of FC dialect to LLVM
- Remaining fc.do loops are lowered to LLVM canonical loops
- OpenMP dialect gets lowered later

LLVM Lowering : fc.array operations

- Array Sections will already be fully lowered to loops and/or array elements
- `fc.array` lowering
    - Contains descriptor structure to hold bounds, size and stride information
    - `LLVM::StructType` to hold array metadata similar to MemRefDescriptor
- `fc.lbound, fc.ubound`
    - Returns lower / upper bound of given array dimension
- Fortran String operations are lowered to strcmp, strcat, etc

LLVM Lowering: Memory operations

- Memory allocation
  - `llvm::AllocaInst` or malloc/free
  - Allocatable arrays use malloc/free
  - Globals are converted to LLVM global variables
  - `AddressOfOp` disappears
- Load and Store operations
  - Lowered to LLVM IR GEP, Load / Store
- `fc.cast_to_memref`
  - Creates std.memref descriptor using fc.array descriptor

LLVM Lowering: IO related operations

- Replace IO operations with runtime library functions

```
fc.print %cst {arg_info = #fc.is_string< >}
```

↓

```
call void (i32, ...) @__fc_runtime_print(..., double 1.110000e+00)
```

OpenMP Lowering:

- OpenMP dialect lowering is scheduled after all other dialects are lowered to LLVM dialect.
- All the OpenMP dialect operations are lowered to LLVM operations.
- Outlines OpenMP region to `LLVM::FuncOp`
- Outlined function is called using OpenMP runtime library function `__kmpc_fork_call(..)`

OpenMP Lowering: Example

```
omp.parallel(%2)  {
  %2 = fc.load %2 {name = "a"} : i32
  fc.print %2  {arg_info =
              #fc.is_string< >}
 }
```

```
define void @outlined.(i32* noalias nocapture readnone %0, i32* noalias
       nocapture readnone %1, i32* noalias nocapture readonly %2) {
  %4 = load i32, i32* %2, align 4, !alias.scope !0, !noalias !2
  tail call void (i32, ...) @__fc_runtime_print(i32 2, ,i32 3, i32 %4)
  ret void
}
define i32 @foo() local_unnamed_addr {
  %1 = alloca i32, align 4
  store i32 10, i32* %1, align 4
  call void ({ i32, i32, i32, i32, i8* }*, i32, i8*, ...)
             @__kmpc_fork_call({ i32, i32, i32, i32, i8* }* nonnull
            @ident.global, i32 1, i8* bitcast (void (i32*, i32*, i32*)*
             @outlined. to i8*), i32* nonnull %1)
  ret i32 0
}
```

LLVM Pass to fix few things!

- LLVM dialect doesn't contain full representation of LLVM IR
- We wrote a custom pass to fix few things
    - `-ffast-math` flags
    - `alias.scope` metadata
    - `LinkageKind` from `mlir::FuncOp` to `LLVM::FuncOp`

Testing (test/)

- Unit tests
  - CTest based
  - Around 400 tests covering all the features implemented
  - Runs parallely.
  - Output are verified with other Fortran compilers (like, gfortran, flang)
  - AST Tests are also included.
- SPEC CPU 2017 benchmarks exchange, bwaves are passing.

Thank You