MLIR Tutorial (Part 2)

Jaeho Lee

Corelab @ Yonsei University

Course Information

Introduction to MLIR

4 Chapters
3 Practices

Based on LLVM 15.0.0

Course Outline

- Frequently used functions
- Generating td files
- Building the Operations
- Conversion Pass

List of Practices

- Frequently used functions
- Generating td files
  - Practice 1: Define Dialect and Operation
- Building the Operations
  - Practice 2: Build the Operation
- Conversion Pass
  - Practice 3: Do a Conversion

Frequently used functions

Frequently used Classes

- Operation
- Value
- OpBuilder
- ModuleOp
- FuncOp
- Type

Member functions of Operation

- **getName()**
  - Retrieves the name of the operation.

```
StringRef name = op.getName().getStringRef();
```

- **getOperands()**
  - Gets all the operands (Value objects) of the operation.

```
OperandRange operands = op.getOperands();
```

Member functions of Operation

- **getResults()**
  - Gets all the results (Value objects) produced by the operation.

```
ResultRange results = op.getResults();
```

- **getAttr(s)**
  - Retrieves one or multiple attributes.

```
Attribute attr = op.getAttr("someAttr");
```

  - Recent version of MLIR

```
Attribute attr = op.getSomeAttr();
```

Member functions of Operation

- **erase()**
  - Deletes the operation.

```
op.erase();
```

How to deal with users

- **getUsers()**
  - Gets an iterator range over the operations that use a specific value.

```
for (Operation* user : value.getUsers()) {
  // Process each user
}
```

Member functions of Operation

How to deal with users

- **hasOneUse()**
  - Checks if a value has only one user.

```
if (value.hasOneUse()) {
  // Do something
}
```

Member functions of Value

- **getType()**
  - Retrieves the type of the value.

```
Type type = value.getType();
```

- **getDefiningOp<T>()**
  - Retrieves the operation that defines this value, cast to a specific type.

```
auto definingOp = value.getDefiningOp<SomeOpType>();
```

Member functions of Value

- **replaceAllUsesWith(otherValue)**
  - Replaces all uses of this value with another value.

```
value.replaceAllUsesWith(otherValue);
```

- **hasOneUse()**
  - Checks if the value is used only once.

```
if (value.hasOneUse()) {
  // Do something
}
```

Member functions of OpBuilder

- **create<OpType>(...)**
  - Creates a new operation of a specific type. Parameters often include result type, operands, and attributes.

```
auto op = builder.create<MyOp>(loc, resultType, operand1, operand2);
```

- **insert(Block *block, Operation *op)**
  - Inserts an operation into a specified block.

```
builder.insert(block, someOperation);
```

Member functions of OpBuilder

- **setInsertionPoint(Operation *op)**
  - Sets the insertion point. New operations will be inserted right after this point.

```
builder.setInsertionPoint(existingOp);
```

- **getI64IntegerAttr(int64_t value)**
  - Creates an integer attribute.

```
auto intAttr = builder.getI64IntegerAttr(1234);
```

Member functions of OpBuilder

- **getStringAttr(StringRef value)**
  - Creates a string attribute.

```
auto strAttr = builder.getStringAttr("hello");
```

Member functions of ModuleOp

- **getOps<T>()**
  - Gets an iterator range of operations of type T within the module.

```
auto funcs = module.getOps<FuncOp>();
for (FuncOp funcOp : funcs) {
 // process each FuncOp Operation in the block.
}
```

- **lookupSymbol<T>(StringRef name)**
  - Look up a symbol by its name.

```
auto func = module.lookupSymbol<FuncOp>("function_name");
```

Member functions of FuncOp

- **getType()**
  - Gets the function type.

```
FunctionType type = func.getType();
```

- **getNumArguments()**
  - Gets the number of arguments for the function.

```
unsigned numArgs = func.getNumArguments();
```

Member functions of Type

- **cast<T>()**
  - Casts the type to a specific subclass.

```
auto floatType = type.cast<FloatType>();
```

- **dyn_cast<T>()**
  - Safely casts to a subclass, returns null if the cast is not possible.

```
auto specificType = type.dyn_cast<SpecificType>();
```

Generating td files

What is “td” files?

- Used to declare new operations, attributes, types, and more
- Automatically generate C++ code

- The **basic** syntax of td files

```
def My_Operation : MyDialect_Op<"my_operation", [/* traits */]> {
  let summary = "Description of my operation";
  let description = "The my_operation does ...";
  let arguments = (ins OperandType:$input1, OperandType:$input2);
  let results = (outs ResultType:$output);
}
```

What is “td” files?

- Used to declare new operations, attributes, types, and more
- Automatically generate C++ code

- The **pro** syntax of td files

```
def My_Operation : MyDialect_Op<"my_operation", [/* traits */]> {
  let summary = "Description of my operation";
  let description = "The my_operation does ...";
  let arguments = (ins OperandType:$input1, OperandType:$input2, I32Attr:$mode);
  let results = (outs ResultType:$output);

  // Builder 1: Takes result type and two input values
  let builders = [
    OpBuilder<(ins "Type":$resultType, "Value":$input1, "Value":$input2)>,

    // Builder 2: Takes an existing operation and an input value
    OpBuilder<(ins "Operation":$existingOp, "Value":$input)>
  ];
}
```

What is “td” files?

```
1  def My_Operation : MyDialect_Op<"my_operation", [/* traits */]> {
2    let summary = "Description of my operation";
3    let description = "The my_operation does ...";
4    let arguments = (ins OperandType:$input1, OperandType:$input2, I32Attr:$mode);
5    let results = (outs ResultType:$output);
6
7    // Builder 1: Takes result type and two input values
8    let builders = [
9      OpBuilder<(ins "Type":$resultType, "Value":$input1, "Value":$input2)>,
10
11     // Builder 2: Takes an existing operation and an input value
12     OpBuilder<(ins "Operation":$existingOp, "Value":$input)>
13   ];
14 }
```

Attribute

Builders

- Define Attributes and Builders
  - Attributes: static information about Op
  - Builders: custom building method of Op

What is “td” files?: Builder

- Explicitly write the Operation building policy

```
def ONNXAddOp:ONNX_Op<"Add",
  [NoSideEffect, DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {
  //...
  let arguments = (ins AnyTypeOf<>:$A, AnyTypeOf<>:$B)
  let results = (outs AnyTypeOf<>:$C)
  let builders = [
    OpBuilder<(ins "Value":$A, "Value":$B), [{
      auto lhsTy = A.getType();
      auto rhsTy = B.getType();
      auto oTy = nullptr;
      auto elementType = getBroadcastedRankedType(lhsTy, rhsTy, oTy);
      auto shapedType = elementType.dyn_cast_or_null<ShapedType>();
      if (!shapedType || !shapedType.hasStaticShape()) {
          elementType = A.getType().cast<ShapedType>().getElementType();
          elementType = UnrankedTensorType::get(elementType);
      }
      build($_builder, $_state, elementType, A, B);
    }]>
    ];
    // ...
}
```

td grammar

C++ grammar

Practice 1: Define Dialect and Operation

- Define Hello dialect
- Define hello.addsign dialect with int32 sign attribute
  - 0: plus / 1: minus

```
(ins F64Tensor:$lhs, F64Tensor:$rhs, I32Attr:$sign);
```

- File hierarchy
  - /root/tutorial/mlir/include/Dialect
  - /root/tutorial/mlir/lib/Dialect
  - /root/tutorial/mlir/toy-opt/CMakeLists.txt
  - toy-opt.cpp
    - #include "Dialect/Toy/IR/ToyOps.hpp"
    - registry.insert<mlir::ToyDialect>();

Building the Operations

Recall: Operation ODS format

- Let's see the example matrix add operation

```
def MatrixAddOp : Op<"matrix.add", [NoSideEffect]> {
  let summary = "Adds two matrices element-wise";

  // Input operands: two matrices of the same type
  let arguments = (ins F32Tensor:$lhs, F32Tensor:$rhs);

  // Output: one matrix of the same type
  let results = (outs F32Tensor:$result);

  // Additional class declarations for C++ code
  let extraClassDeclaration = [{
    // You can add C++ methods here for better code management
  }];

  // Example assembly format for this operation
  let assemblyFormat = "tensor-type(operands) `:` type(operands)";
}
```

Operation: Verifier

- How to verify the two arguments are possible to add?

```
def MatrixAddOp : Op<"matrix.add", [NoSideEffect]> {
  let summary = "Adds two matrices element-wise";

  // Input operands: two matrices of the same type
  let arguments = (ins F32Tensor:$lhs, F32Tensor:$rhs);

  // Output: one matrix of the same type
  let results = (outs F32Tensor:$result);

  // Additional class declarations for C++ code
  let extraClassDeclaration = [{
    // You can add C++ methods here for better code management
  }];

  // Example assembly format for this operation
  let assemblyFormat = "tensor-type(operands) `:` type(operands)";
}
```

Operation: Verifier

- /root/tutorial/mlir/lib/Dialect/Toy/IR/ToyOps.cpp

```
//===----------------------------------------------------------------------===//
// AddOp

void AddOp::build(mlir::OpBuilder& builder, mlir::OperationState& state,
    mlir::Value lhs, mlir::Value rhs)
{
  state.addTypes(UnrankedTensorType::get(builder.getF64Type()));
  state.addOperands({ lhs, rhs });
}
//===----------------------------------------------------------------------===//
// SubOp

void SubOp::build(mlir::OpBuilder& builder, mlir::OperationState& state,
    mlir::Value lhs, mlir::Value rhs)
{
  state.addTypes(UnrankedTensorType::get(builder.getF64Type()));
  state.addOperands({ lhs, rhs });
}
```

Operation: Verifier

- Define the verify function

```
LogicalResult MatrixAddOp::verify() {
  // Verify code here, for example:
  if (lhs().getType() != rhs().getType()) {
    return emitOpError("lhs and rhs should have the same type");
  }

  return success();
}
```

Operation: OpBuilder

- Define Opbuilder

```
OpBuilder builder(&getContext());
```

- Define loc

```
auto loc = moduleOp.getLoc();
```

Operation: OpBuilder

- Set insertion point

```
builder.setInsertionPointAfter(op);
```

```
builder.setInsertionPoint(op);
```

- Define loc

```
builder.create<AddOp>(loc, op->getResult(0).getType(),
                      op->getResult(0), user->getResult(0));
```

Operation: OpBuilder

- Example of Building Operation

```
module {
  func @main(%arg0: tensor<2x3xf64>, %ar
    %0 = "toy.add"(%arg0, %arg1) : (tens
    %1 = "toy.sub"(%0, %arg1) : (tensor<
    %2 = "toy.add"(%0, %1) : (tensor<2x3
    %3 = "toy.add"(%1, %arg1) : (tensor<
    toy.return %3 : tensor<2x3xf64>
  }
}
```

```
30 void removeRedundantAddSubPass::runOnOperation() {
31   OpBuilder builder(&getContext());
32   auto loc = moduleOp.getLoc();
33   auto moduleOp = getOperation();
34
35   moduleOp.walk([&](Operation *op) {
36     if (isa<AddOp>(op)) {
37       for (Operation* user: op->getUsers()) {
38         if (isa<SubOp>(user)) {
39           auto addArg1 = op->getOperand(1);
40           auto subArg1 = user->getOperand(1);
41           if (addArg1 == subArg1) {
42             builder.setInsertionPointAfter(user);
43             builder.create<AddOp>(loc, op->getResult(0).getType(),
44                 op->getResult(0), user->getResult(0));
45           }
46         }
47       }
48     }
49   });
50 }
```

Practice 2: Build the Operation

- Build the Operation and replace the use!

```
module {
  func.func @main(%arg0, %arg1]) {
    %0 = "toy.add"(%arg0, %arg1)
    %1 = "toy.sub"(%0, %arg1)
    %2 = "toy.add"(%1, %arg1)
    toy.return %2
  }
}
```

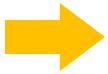
```
module {
  func.func @main(%arg0, %arg1]) {
    %0 = "toy.add"(%arg0, %arg1)
    %1 = "toy.sub"(%0, %arg1)
    %2 = "toy.add"(%1, %arg1)
    %3 = "toy.sub"(%0, %2)
    toy.return %3
  }
}
```

Conversion Pass

What is Conversion?

- Change the code into another dialect

```
0  h_A = INPUT
1  h_B = onnx.Constant()
2  h_C = onnx.Constant()
3  h_D = onnx.Conv(h_A,h_B,h_C)
4  h_E = onnx.Relu(h_D)
5  h_F = onnx.Constant()
6  h_G = onnx.Add(h_E,h_F)
```



```
 0  h_A = INPUT
 1  d_A = DNN Malloc(768)
 2  t0  = DNN Memcpy(d_A,h_A) {H->D}
 3  d_B = onnx.Constant()
 4  h_B = DNN Malloc(540)
 5  t1  = DNN Memcpy(d_D,h_D) {H->D}
 6  h_C = onnx.Constant()
 7  d_C = DNN Malloc(20)
 8  t2  = DNN Memcpy(d_C,h_C) {H->D}
 9  d_D = DNN Malloc(720)
10  t3  = DNN Conv(d_A,d_B,d_C,d_D)
11  t4  = DNN Dealloc(d_A)
12  t5  = DNN Dealloc(d_B)
13  t6  = DNN Dealloc(d_C)
14  d_E = DNN Malloc(720)
15  t7  = DNN Relu(d_D,d_E)
16  t8  = DNN Dealloc(d_D)
17  h_F = onnx.Constant()
18  d_F = DNN Malloc(720)
19  t9  = DNN Memcpy(d_F,h_F) {H->D}
20  d_G = DNN Malloc(720)
21  t10 = DNN Add(d_E,d_F,d_G)
22  t11 = DNN Dealloc(d_E)
23  t12 = DNN Dealloc(d_F)
24  h_G = DNN Host-malloc()
25  t13 = DNN Memcpy(h_G,d_G) {D->H}
26  t14 = DNN Dealloc(d_G)
```

What is Conversion?

- Conversion in MLIR is basically a **pattern matching**

```
//===----------------------------------------------------------------------===//
// AddOp
//===----------------------------------------------------------------------===//
struct ToyAddOpToHello : public mlir::ConversionPattern {
  ToyAddOpToHello(MLIRContext* context)
    : ConversionPattern(mlir::AddOp::getOperationName(), 1, context)
  {
  }

  LogicalResult matchAndRewrite(mlir::Operation* op, mlir::ArrayRef<Value> operands,
      mlir::ConversionPatternRewriter& rewriter) const final
  {
    // Do Somthing
    return success();
  }
};

void mlir::populateLoweringToyAddOpToHelloPatterns(
    RewritePatternSet& patterns, MLIRContext* context)
{
  patterns.insert<ToyAddOpToHello>(context);
}
```

Practice 3: Do a Conversion

- Convert toy.add and toy.sub operations
  - Into hello.addsign operations

```
module {
  func.func @main(%arg0, %arg1]) {
    %0 = "toy.add"(%arg0, %arg1)
    %1 = "toy.sub"(%0, %arg1)
    %2 = "toy.add"(%1, %arg1)
    toy.return %2
  }
}
```

```
module {
  func.func @main(%arg0, %arg1]) {
    %0 = "hello.addsign"(%arg0, %arg1, 0)
    %1 = "hello.addsign"(%0, %arg1, 1)
    %2 = "hello.addsign"(%1, %arg1, 0)
    toy.return %2
  }
}
```