MLIR

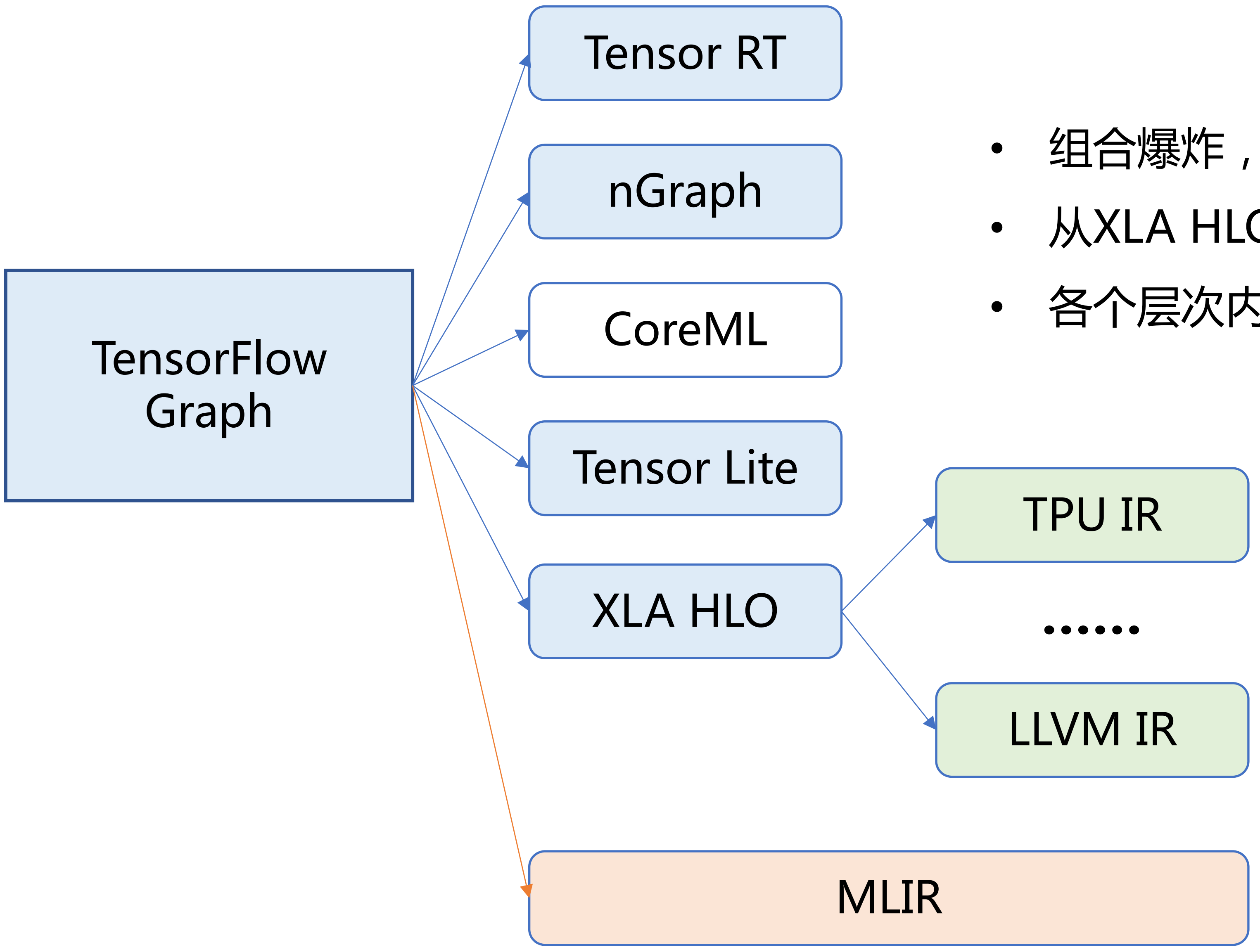
Multi-Level IR Compiler Framework

MLIR Toy Tutorial 概述

张洪滨

2020.02.26

前情提要



- 组合爆炸，很多组件无法重用
- 从XLA HLO到LLVM IR跨度太大，实现开销大
- 各个层次内部优化无法迁移

- 统一IR，重用组件，避免组合爆炸
- 采用多级抽象， Multi-Level的真意
- 各个层次进行协同优化

Toy Tutorial 目标效果

通过展示Toy语言的编译流程，介绍MLIR的概念。

This tutorial runs through the implementation of a basic toy language on top of MLIR. The goal of this tutorial is to introduce the concepts of MLIR; in particular, how dialects can help easily support language specific constructs and transformations while still offering an easy path to lower to LLVM or other codegen infrastructure.

具体来说，Toy Tutorial 展示Dialect是如何在Toy语言的分析、表达式变型、Lowering到LLVM的过程中发挥作用。

Toy Tutorial 章节划分

- Chapter 1 – 介绍Toy语言以及抽象语法树AST
- Chapter 2 – 定义Toy Dialect，Toy Operation，生成MLIR表达式
- Chapter 3 – Toy Operation层级的表达式变型
- Chapter 4 – 使用接口，完成泛化的表达式变型
- Chapter 5 – 将MLIR表达式进行部分Lowering，并进行优化
- Chapter 6 – 混合Dialect表达式Lowering到LLVM IR
- Chapter 7 – 扩展源语言，向Toy语言添加struct数据类型

添加Dialect描述
分析AST
生成MLIR表达式

当MLIR表达式存在冗余
进行针对Operation的表达式变型

使用已有接口
完成泛化的表达式变型

将Toy Dialect 的部分Operation
映射到Affine Dialect Operation
对Affine MLIR表达式进行优化

混合Dialect MLIR表达式
-> LLVM IR Dialect MLIR表达式
-> LLVM IR 表达式

在现有的编译流程中
添加自定义的数据类型

定义转置相乘的函数

```
def multiply_transpose(a, b) {
  return transpose(a) * transpose(b);
}
```

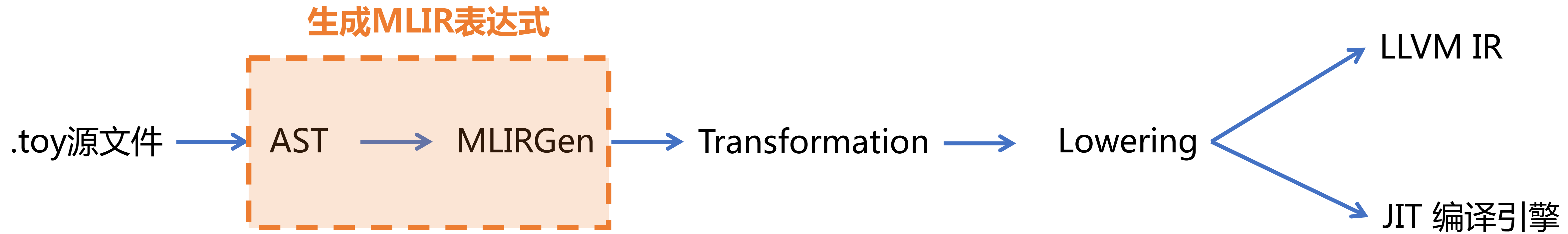
张量对应元素相乘

main函数

Reshape操作

```
def main() {
  var a<2, 3> = [[1, 2, 3], [4, 5, 6]];
  var b<2, 3> = [1, 2, 3, 4, 5, 6];
  var c = multiply_transpose(a, b);
  var d = multiply_transpose(b, a);
  print(d);
}
```

打印计算结果



MLIRGen模块

遍历AST递归调用子函数

根据不同的输入类型调用相应子函数

```
return builder.create<TransposeOp>(location, operands[0]);
```

Dialect模块

负责定义各种操作和分析，具备可扩展性。

```
void TransposeOp::build(mlir::Builder *builder, mlir::OperationState &state,
                        mlir::Value value) {
  state.addTypes(UnrankedTensorType::get(builder->getF64Type()));
  state.addOperands(value);
}
```

Operation模块

定义各种操作的类

在编译时向Dialect模块提供支持

```
def TransposeOp : Toy_Op<"transpose"> {
  ... ...
}
```

Operation模块

定义各种操作的类

在编译时向Dialect模块提供支持

```
def TransposeOp : Toy_Op<"transpose"> {
  let summary = "transpose operation";

  let arguments = (ins F64Tensor:$input);
  let results = (outs F64Tensor);

  // Allow building a TransposeOp with from the input operand.
  let builders = [
    OpBuilder<"Builder *b, OperationState &state, Value input">
  ];

  // Invoke a static verify method to verify this transpose operation.
  let verifier = [{ return ::verify(*this); }];
}
```

添加文档

使用summary或者description字段，给Operation提供文档支持。提供的信息可以自动生成Markdown文档。

定义输入参数和输出结果

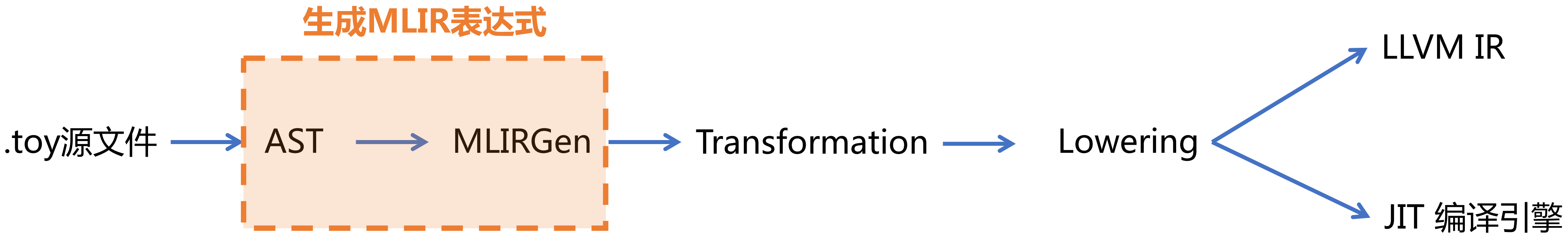
arguments和results字段相当于Operation的输入和输出，输入的参数是基于SSA的操作数类型。输出的结果是Operation产生的值的类型。

定义build方法

ODS框架会自定义一些build方法，在builders字段中，可以自定义一个OpBuilder对象列表，其中用字符串的形式定义参数。

Operation语义验证

验证函数会自动生成，为了添加自定义的验证内容，使用verifier字段，将自定义的验证内容添加到自动生成的内容之后。



相同的维数进行reshape操作，存在冗余

```
module {
  func @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
    %0 = "toy.transpose"(%arg0) : (tensor<*xf64>) -> tensor<*xf64>
    %1 = "toy.transpose"(%arg1) : (tensor<*xf64>) -> tensor<*xf64>
    %2 = "toy.mul"(%0, %1) : (tensor<*xf64>, tensor<*xf64>) -> tensor<*xf64>
    "toy.return"(%2) : (tensor<*xf64>) -> ()
  }
  func @main() {
    %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00,
                                            6.000000e+00]]> : tensor<2x3xf64>} : () -> tensor<2x3xf64>
    %1 = "toy.reshape"(%0) : (tensor<2x3xf64>) -> tensor<2x3xf64>
    %2 = "toy.constant"() {value = dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00,
                                           6.000000e+00]> : tensor<6xf64>} : () -> tensor<6xf64>
    %3 = "toy.reshape"(%2) : (tensor<6xf64>) -> tensor<2x3xf64>
    %4 = "toy.generic_call"(%1, %3) {callee = @multiply_transpose} : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %5 = "toy.generic_call"(%3, %1) {callee = @multiply_transpose} : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    "toy.print"(%5) : (tensor<*xf64>) -> ()
    "toy.return"() : () -> ()
  }
}
```

MLIRGen模块

遍历AST递归调用子函数
根据不同的输入类型调用相应子函数

```
return builder.create<TransposeOp>(location, operands[0]);
```

PassManger模块

使用PassManger添加一道优化工序
运行定义好的canonicalizer用来优化MLIR模型

```
if (enableOpt) {
  mlir::PassManager pm(&context);
  ......
  // Add a run of the canonicalizer to optimize the mlir module.
  pm.addNestedPass<mlir::FuncOp>(mlir::createCanonicalizerPass());
  if (mlir::failed(pm.run(*module)))
    return 4;
}
```

Transformation模块

定义匹配和重写模式
实现MLIR模型的优化

方法一：采用C++直接编写matchAndRewrite函数

```
matchAndRewrite(TransposeOp op,
mlir::PatternRewriter &rewriter) const override {
  ......
}
```

方法二：采用DRR，自动生成匹配和重写函数

```
def ReshapeReshapeOptPattern :
Pat<(ReshapeOp(ReshapeOp $arg)), (ReshapeOp $arg)>;
```

将自定义的匹配和重写模式登记为canonicalization模式

```
void TransposeOp::getCanonicalizationPatterns(......) {
  results.insert<SimplifyRedundantTranspose>(context);
}
```


```
module {
  func @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
    %0 = "toy.transpose"(%arg0) : (tensor<*xf64>) -> tensor<*xf64>
    %1 = "toy.transpose"(%arg1) : (tensor<*xf64>) -> tensor<*xf64>
    %2 = "toy.mul"(%0, %1) : (tensor<*xf64>, tensor<*xf64>) -> tensor<*xf64>
    "toy.return"(%2) : (tensor<*xf64>) -> ()
  }
  func @main() {
    %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00,
                                           6.000000e+00]]> : tensor<2x3xf64>} : () -> tensor<2x3xf64>
    %1 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00,
                                           6.000000e+00]]> : tensor<2x3xf64>} : () -> tensor<2x3xf64>
    %2 = "toy.generic_call"(%0, %1) {callee = @multiply_transpose} : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %3 = "toy.generic_call"(%1, %0) {callee = @multiply_transpose} : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    "toy.print"(%3) : (tensor<*xf64>) -> ()
    "toy.return"() : () -> ()
  }
}
```

MLIR表达式中去掉冗余reshape操作

为了代码执行速度快，将函数进行内联操作
为了代码生成阶段方便，需要确定所有的tensor形状

PassManger模块

使用PassManger添加优化工序

```
if (enableOpt) {
  mlir::PassManager pm(&context);
  ......
  // Inline all functions into main
  // and then delete them.
  pm.addPass(mlir::createInlinerPass());
  pm.addPass(mlir::createSymbolDCEPass());

  // Now that there is only one function,
  // we can infer the shapes of each of
  // the operations.
  mlir::OpPassManager &optPM = pm.nest<mlir::FuncOp>();
  optPM.addPass(mlir::toy::createShapeInferencePass());
  optPM.addPass(mlir::createCanonicalizerPass());
  optPM.addPass(mlir::createCSEPass());

  ......
}
```

内联 Pass

1. 继承内联接口，实现变型规则
2. 定位函数调用位置
3. 将变量类型转变为函数参数类型

Shape推断 Pass

1. 使用ODS框架生成Shape推断接口类
2. 在Operation模块中添加接口
3. 定义各Operation的Shape推断函数
4. 编写Shape推断 Pass

PassManger模块

使用PassManger添加优化工序

```
if (enableOpt) {
  mlir::PassManager pm(&context);
  ......
  // Inline all functions into main
  // and then delete them.
  pm.addPass(mlir::createInlinerPass());
  pm.addPass(mlir::createSymbolDCEPass());
  ......
}
```

内联 Pass

1. 继承内联接口，实现变型规则

2. 定位函数调用位置

3. 将变量类型转变为函数参数类型

Dialect模块

ToyInlinerInterface 继承 DialectInlinerInterface
实现基类中的相应函数，启用内联，定义表达式变型规则

```
struct ToyInlinerInterface : public DialectInlinerInterface {
  bool isLegalToInline(...) const final {
    return true;
  }
  void handleTerminator(...) const final {...}
}
```

登记 ToyInlinerInterface

```
addInterfaces<ToyInlinerInterface>();
```

实现 GenericCallOp 成员函数，返回被调用函数，并获取参数操作数

```
CallInterfaceCallable GenericCallOp::getCallableForCallee() {...}
Operation::operand_range GenericCallOp::getArgOperands() {...}
```

Operation模块

识别函数调用位置，在GenericCallOp中添加接口

```
def GenericCallOp : Toy_Op<"generic_call",
[DeclareOpInterfaceMethods<CallOpInterface>]> { ... }
```

PassManger模块

使用PassManger添加优化工序

```
if (enableOpt) {
  mlir::PassManager pm(&context);
  ......
  // Inline all functions into main
  // and then delete them.
  pm.addPass(mlir::createInlinerPass());
  pm.addPass(mlir::createSymbolDCEPass());
  ......
}
```

内联 Pass

1. 继承内联接口，实现变型规则
2. 定位函数调用位置
3. 将变量类型转变为函数参数类型

函数定义时，泛化的tensor类型

```
func @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>)
-> tensor<*xf64>
```

函数调用时，shape确定的tensor类型

内联时需要统一类型

```
%4 = "toy.generic_call"(%1, %3) {callee = @multiply_transpose} :
(tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
```

Operation模块

添加CastOp，将一个tensor转换成等价的不同shape的tensor

```
def CastOp : Toy_Op<"cast",
    [DeclareOpInterfaceMethods<ShapeInferenceOpInterface>, NoSideEffect,
     SameOperandsAndResultShape]> {...}
```

Dialect模块

在内联接口中，重写相应函数，构造CastOp表达式

```
struct ToyInlinerInterface : public DialectInlinerInterface {
  Operation *materializeCallConversion(...) const final {
    return builder.create<CastOp>(conversionLoc, resultType, input);
  }
}
```

Shape推断 Pass

1. 使用ODS框架生成Shape推断接口类
2. 在Operation模块中添加接口
3. 定义各Operation的Shape推断函数
4. 编写Shape推断 Pass

Shape推断接口模块

编写Shape推断的tablegen文件，使用ODS框架生成代码

```
def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {
  ......
}
```

Operation模块

向需要Shape推断的Operation添加接口

```
def MulOp : Toy_Op<"mul", [NoSideEffect,
DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {...}
def TransposeOp : Toy_Op<"transpose", [NoSideEffect,
DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {...}
```

Dialect模块

定义各Operation的Shape推断函数

```
void MulOp::inferShapes() { getResult().setType(getOperand(0).getType()); }
void TransposeOp::inferShapes() {
  auto arrayTy = getOperand().getType().cast<RankedTensorType>();
  SmallVector<int64_t, 2> dims(llvm::reverse(arrayTy.getShape()));
  getResult().setType(RankedTensorType::get(dims, arrayTy.getElementType()));
}
```

PassManger模块

使用PassManger添加优化工序

```
if (enableOpt) {
  ......
  mlir::OpPassManager &optPM = pm.nest<mlir::FuncOp>();
  optPM.addPass(mlir::toy::createShapeInferencePass());
  optPM.addPass(mlir::createCanonicalizerPass());
  optPM.addPass(mlir::createCSEPass());
  ......
}
```

Shape推断 Pass

1. 使用ODS框架生成Shape推断接口类
2. 在Operation模块中添加接口
3. 定义各Operation的Shape推断函数
4. 编写Shape推断 Pass

Shape推断Pass 模块

定义一个Shape推断Pass类，实现Shape推断算法

```
class ShapeInferencePass : public mlir::FunctionPass<ShapeInferencePass> {
  ......
};
```

创建一个Shape推断的Pass

```
std::unique_ptr<mlir::Pass> mlir::toy::createShapeInferencePass() {
  return std::make_unique<ShapeInferencePass>();
}
```


```
module {
  func @main() {
    %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00,
                              5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>} : () -> tensor<2x3xf64>
    %1 = "toy.transpose"(%0) : (tensor<2x3xf64>) -> tensor<3x2xf64>
    %2 = "toy.mul"(%1, %1) : (tensor<3x2xf64>, tensor<3x2xf64>) -> tensor<3x2xf64>
    "toy.print"(%2) : (tensor<3x2xf64>) -> ()
    "toy.return"() : () -> ()
  }
}
```

完成内联以及shape推断操作

将MLIR表达式向下Lowering，并进行表达式优化

PassManger模块

使用PassManger添加优化工序

```
if (isLoweringToAffine) {
  // Partially lower the toy dialect with a few cleanups afterwards.
  pm.addPass(mlir::toy::createLowerToAffinePass());

  mlir::OpPassManager &optPM = pm.nest<mlir::FuncOp>();
  optPM.addPass(mlir::createCanonicalizerPass());
  optPM.addPass(mlir::createCSEPass());

  // Add optimizations if enabled.
  if (enableOpt) {
    optPM.addPass(mlir::createLoopFusionPass());
    optPM.addPass(mlir::createMemRefDataFlowOptPass());
  }
}
```

ToyToAffineLoweringPass

1. 匹配重写Operation pattern
2. 添加target和pattern

LoopFusionPass

MemRefDataFlowOptPass

消除冗余的load操作

使得转置和矩阵相乘在一个循环中完成

PassManger模块

使用PassManger添加优化工序

```
if (isLoweringToAffine) {
  // Partially lower the toy dialect
  // with a few cleanups afterwards.
  pm.addPass(mlir::toy::createLowerToAffinePass());

  ......
}
```

ToyToAffineLoweringPass

1. 匹配重写Operation pattern
2. 添加target和pattern

ToyToAffine模块

定义Operation Lowering
匹配Toy中的Operation，并且进行Operation重写

```
struct TransposeOpLowering : public ConversionPattern {...}
```

添加Affine和Standard Dialect target
将原有Toy Dialect设为非法target
添加Operation Lowering pattern

```
void ToyToAffineLoweringPass::runOnFunction() {
  ......
  target.addLegalDialect<AffineOpsDialect, StandardOpsDialect>();
  target.addIllegalDialect<toy::ToyDialect>();
  target.addLegalOp<toy::PrintOp>();
  OwningRewritePatternList patterns;
  patterns.insert<AddOpLowering, ConstantOpLowering, MulOpLowering,
  ReturnOpLowering, TransposeOpLowering>(&getContext());
  ......
}
```

PassManger模块

使用PassManger添加优化工序

```
if (isLoweringToAffine) {
  ......
  if (enableOpt) {
    optPM.addPass(mlir::createLoopFusionPass());
    optPM.addPass(mlir::createMemRefDataFlowOptPass());
  }
}
```

LoopFusionPass

MemRefDataFlowOptPass

消除冗余的load操作

使得转置和矩阵相乘在一个循环中完成

```
affine.for %arg0 = 0 to 3 {
  affine.for %arg1 = 0 to 2 {
    %3 = affine.load %2[%arg1, %arg0] : memref<2x3xf64>
    affine.store %3, %1[%arg0, %arg1] : memref<3x2xf64>
  }
}
affine.for %arg0 = 0 to 3 {
  affine.for %arg1 = 0 to 2 {
    %3 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>
    %4 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>
    %5 = mulf %3, %4 : f64
    affine.store %5, %0[%arg0, %arg1] : memref<3x2xf64>
  }
}
```

```
affine.for %arg0 = 0 to 3 {
  affine.for %arg1 = 0 to 2 {
    %2 = affine.load %1[%arg1, %arg0] : memref<2x3xf64>
    %3 = mulf %2, %2 : f64
    affine.store %3, %0[%arg0, %arg1] : memref<3x2xf64>
  }
}
```


```
module {
  func @main() {
    %cst = constant 1.000000e+00 : f64
    %cst_0 = constant 2.000000e+00 : f64
    %cst_1 = constant 3.000000e+00 : f64
    %cst_2 = constant 4.000000e+00 : f64
    %cst_3 = constant 5.000000e+00 : f64
    %cst_4 = constant 6.000000e+00 : f64
    %0 = alloc() : memref<3x2xf64>
    %1 = alloc() : memref<2x3xf64>
    affine.store %cst, %1[0, 0] : memref<2x3xf64>
    affine.store %cst_0, %1[0, 1] : memref<2x3xf64>
    affine.store %cst_1, %1[0, 2] : memref<2x3xf64>
    affine.store %cst_2, %1[1, 0] : memref<2x3xf64>
    affine.store %cst_3, %1[1, 1] : memref<2x3xf64>
    affine.store %cst_4, %1[1, 2] : memref<2x3xf64>
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %2 = affine.load %1[%arg1, %arg0] : memref<2x3xf64>
        %3 = mulf %2, %2 : f64
        affine.store %3, %0[%arg0, %arg1] : memref<3x2xf64>
      }
    }
```

混合Dialect的MLIR表达式需要进一步Lowering

```
    "toy.print"(%0) : (memref<3x2xf64>) -> ()
    dealloc %1 : memref<2x3xf64>
    dealloc %0 : memref<3x2xf64>
    return
  }
}
```

语法分析 **表达式生成** **将函数定义与函数调用内联**

.toy源文件 → AST → MLIRGen → Transformation → Lowering → LLVM IR / JIT 编译引擎

Tensor Shape推断

PassManger模块
使用PassManger添加优化工序

混合Dialect表达式Lowering到LLVM IR

```
if (isLoweringToLLVM) {
  // Finish lowering the toy IR to the LLVM dialect.
  pm.addPass(mlir::toy::createLowerToLLVMPass());
}
```

createLowerToLLVMPass

- 确定Lowering target
- 将MemRef类型映射到LLVM IR Dialect的表达式
- 定义Lowering pattern
- 进行完全Lowering

LLVM IR Dialect

MLIR表达式

将MLIR表达式翻译成LLVM IR表达式再进行LLVM IR的优化处理

定义MLIR的执行引擎
`mlir::ExecutionEngine`

混合Dialect表达式Lowering到LLVM IR

LLVM IR Dialect
MLIR表达式

```
module {
  llvm.func @free(!llvm<"i8*">)
  llvm.mlir.global internal constant @nl("\0A\00")
  llvm.mlir.global internal constant @frmt_spec("%f \00")
  llvm.func @printf(!llvm<"i8*">, ...) -> !llvm.i32
  llvm.func @malloc(!llvm.i64) -> !llvm<"i8*">
  llvm.func @main() {
    %0 = llvm.mlir.constant(1.000000e+00 : f64) : !llvm.double
    %1 = llvm.mlir.constant(2.000000e+00 : f64) : !llvm.double
    %2 = llvm.mlir.constant(3.000000e+00 : f64) : !llvm.double
  ......
}
```

JIT编译引擎
执行结果

```
1.000000 16.000000
4.000000 25.000000
9.000000 36.000000
```

LLVM IR表达式

```
define void @main() local_unnamed_addr #1 {
.preheader3:
  %0 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]*
                                @frmt_spec, i64 0, i64 0), double 1.000000e+00)
  %1 = tail call i32 (i8*, ...) @printf(i8* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]*
                                @frmt_spec, i64 0, i64 0), double 1.600000e+01)
  %putchar = tail call i32 @putchar(i32 10)
  ......
}
```

扩展源语言

.toy源文件 → AST → MLIRGen → Transformation → Lowering → LLVM IR / JIT 编译引擎

将函数定义与函数调用内联

Tensor Shape推断

Partial Lowering

Full Lowering

1 定义Type 类

2 语法分析&打印

3 StructType上的操作

Dialect模块

在ToyTypes命名空间中，添加struct

```
namespace ToyTypes {
  enum Types {
    Struct = mlir::Type::FIRST_TOY_TYPE,
  };
}
```

定义存储类

```
struct StructTypeStorage : public mlir::TypeStorage {...}
```

定义用户可见的StructType

```
class StructType : public mlir::Type::TypeBase<StructType,
mlir::Type, detail::StructTypeStorage> {...}
```

1 定义Type 类

2 语法分析&打印

3 StructType上的操作

Dialect模块

重载parseType函数，对struct语法结构进行语法分析

```
mlir::Type ToyDialect::parseType(mlir::DialectAsmParser &parser) const {
  ...
}
```

重载printType函数，打印struct语法结构

```
void ToyDialect::printType(mlir::Type type,
                           mlir::DialectAsmPrinter &printer) const {
  ...
}
```

扩展源语言

.toy源文件 → AST → MLIRGen → Transformation → Lowering → LLVM IR / JIT 编译引擎

将函数定义与函数调用内联

Tensor Shape推断

Partial Lowering

Full Lowering

1 定义Type 类

2 语法分析&打印

3 StructType上的操作

Operation模块

升级现有的Operation定义，使其能够使用struct数据类型

```
def ReturnOp : Toy_Op<"return", [Terminator, HasParent<"FuncOp">]> {
  ...
  let arguments = (ins Variadic<Toy_Type>:$input);
  ...
}
```

添加Operation，针对性地处理struct数据类型

```
def StructConstantOp : Toy_Op<"struct_constant", [NoSideEffect]> {...}
def StructAccessOp : Toy_Op<“struct_access”, [NoSideEffect]> {...}
```

ToyCombine模块

对 struct Operation进行优化

```
OpFoldResult StructConstantOp::fold(ArrayRef<Attribute> operands) {...}
OpFoldResult StructAccessOp::fold(ArrayRef<Attribute> operands) {...}
```

带有struct数据类型的源程序

```
struct Struct {
  var a;
  var b;
}

def multiply_transpose(Struct value) {
  return transpose(value.a) * transpose(value.b);
}

def main() {
  Struct value = {[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]};

  var c = multiply_transpose(value);
  print(c);
}
```

JIT编译引擎
执行结果

```
1.000000 16.000000
4.000000 25.000000
9.000000 36.000000
```

未来工作

- ~~完成MLIR Toy的学习路线~~
- 调研各种python binding，比较哪种更适合项目
- 学习Operation Definition Specification (ODS)框架

相关文章

- MLIR ODS 框架的使用示例 -- 自定义Operation：https://zhuanlan.zhihu.com/p/105576276
- MLIR 表达式变型：https://zhuanlan.zhihu.com/p/105905654
- MLIR 实现泛化的表达式变型：https://zhuanlan.zhihu.com/p/106472878
- MLIR 表达式优化 -- 部分Lowering：https://zhuanlan.zhihu.com/p/107137298
- MLIR 表达式Lowering到LLVM IR：https://zhuanlan.zhihu.com/p/108386819
- MLIR 向源语言添加struct类型：https://zhuanlan.zhihu.com/p/108575517