# MLIR+TVM Demo简介

<https://github.com/MingliSun/MLIR-TVM/tree/master/Ch8>

本项目为融合MLIR和TVM的一个简单demo，前端采用MLIR教程的Toy语言(一种C风格的张量编程语言)，对Toy源码进行词法分析、语法分析（递归下降）、语义检查，生成MLIR的中间代码Toy Dialect。为了面向TVM生成代码，我们自行设计了Relay Dialect，设计并实现了从Toy Dialect到Relay Dialect的转化机制，并面向Relay Dialect中间代码实现了一个简单的CodeGen代码生成器，可以生成含有Relay API的Python代码，该代码可以被TVM接受，并正确运行。目前实现的系统架构如图1所示。

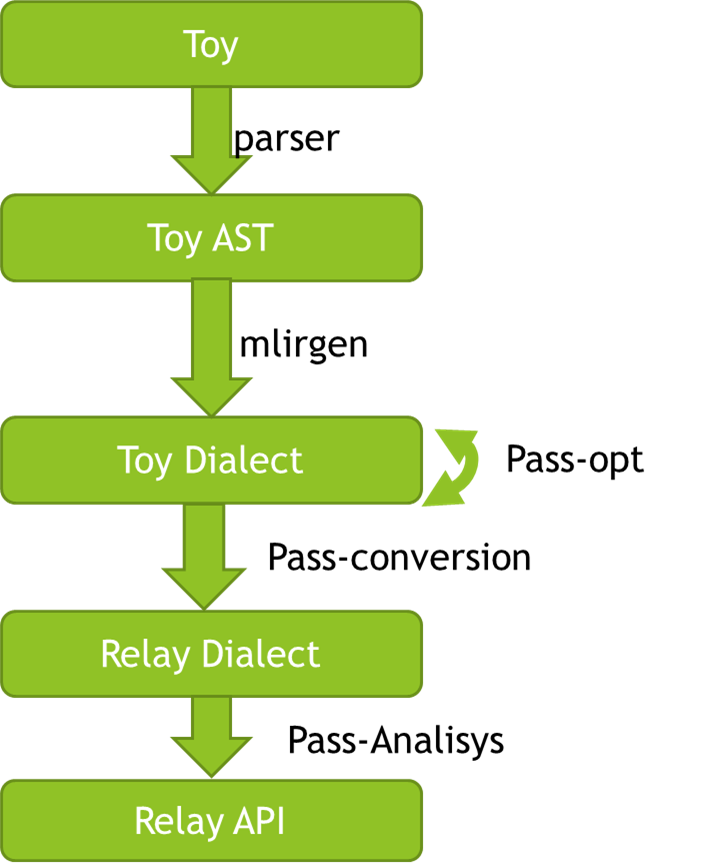


图1. TOY+MLIR+RELAY+TVM编译器架构图

MLIR在表达中间代码时中采用了一种被称为Dialect（方言）的概念，Toy源码经编译生成的MLIR中间代码被称为Toy Dialect。我们可以在同一Dialect中进行优化，MLIR已经实现了一定数量的基本优化，例如，Shape Inference, inlining, DeadFunctionElimination, CSE等。

接下来我们利用已经优化的Toy Dialect转化成Relay Dialect。在这里Toy Dialect和Relay Dialect 抽象程度几乎一致，所以看上去是一个平行转换。得到Relay Dialect后写一个分析pass 来实现Relay Dialect到 Relay API的转化，也可称为Relay CodeGen代码生成。针对Demo提供的测试用例，生成的包含Relay API的Python代码，可以被TVM接受，并正确运行。

以下为几点说明：

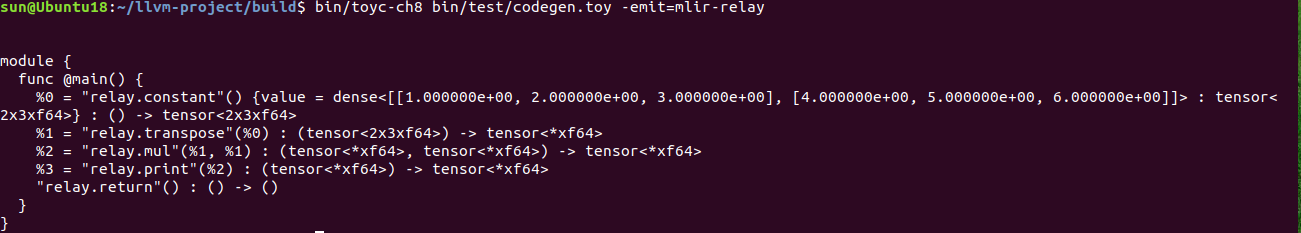
1. Toy Dialect的设计是和Toy语言对应的，包括ContantOp,AddOp,CastOp,GenericCallOp,MulOp,PrintOp,ReshapeOp,ReturnOp,TransposeOp等。Toy语言是一种基于tensor运算的类C语言，内置加法和乘法，内置函数只有print和transpose，能够实现函数调用。
2. Relay API 是 Relay提供的python API，Relay Dialect主要是对每一条Relay API提供方言描述（类似LLVM dialect 和 LLVM IR），在此demo中只实现了部分能用到的Operation。
3. 目前Relay Dialect 也有ContantOp, AddOp, CastOp, GenericCallOp, MulOp, PrintOp, ReshapeOp, ReturnOp, TransposeOp等，与Toy dialect是一对一的关系。
4. 关于Toy是类C语言的程序，Relay是”先build后run”风格的程序,我们目前采用的是print驱动的方法，即程序没有遇到print时将Relay Dialect的Operation 一一对应转换，遇到print，构造函数（以所有常量为输入（feature），print的参数为输出(label)），建立模型，模型的特征输入取常数值(constant生成)，并取出模型的训练结果（predictions）作为输出。例如，针对图2所示的Relay Dialect中间代码：

图2. Relay Dialect中间代码示例

在分析到Relay.Print之前，CodeGen生成代码主要完成从中间代码到目标代码的一一转换，当分析道print时，将按照下列模式建立Relay Function和Module:

***f1 = relay.Function([x],z)***

***mod = relay.Module.from\_expr(f1)***

***mod = relay.transform.InferType()(mod)***

***```***

***module.set\_input("x", data)***

***module.run()***

***out = module.get\_output(0).asnumpy()***

***print(out)***

1. 下面我们通过一个完整的例子说明目前系统的功能。有Toy代码如下：

经过编译后，生成的Toy Dialect中间代码如下：

# User defined generic function that operates on unknown shaped arguments.

def multiply\_transpose(a, b) {

return transpose(a) \* transpose(b);

}

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);

}

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>

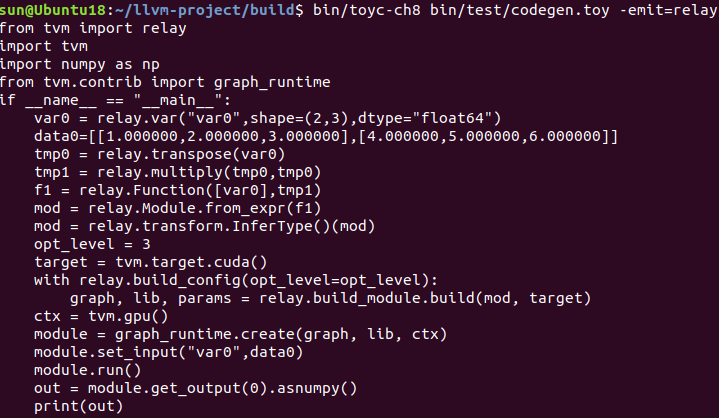
"toy.print"(%5) : (tensor<\*xf64>) -> ()

"toy.return"() : () -> ()

}

}

经过MLIR提供的诸如常数传播，inline，CSE等优化之后，我们将其转化为自定义的Relay Dialect，如图2所示。最后通过Relay CodeGen生成的Realy API Python代码如下：



上述代码运行后得到如下运行结果：

