MLIR 编译框架的使用与探索
第三部分：代码优化

Junhao Dai daijunhao@sjtu.edu.cn

June 23, 2024

1 简介

1.1 实验目的

本次实验旨在探索并使用编译基础设施框架 MLIR(Multi-Level Intermediate Representation)，利用课堂所学习的词法分析、语法分析以及代码优化与生成的知识，去完善基于张量的 Pony 语言，从编译的角度去解析 Pony 语言。第三部分中我们需要实现代码优化操作中的消除 Pony 程序中冗余的 transpose 函数，以及中间代码生成中将 Pony 语言的二维矩阵乘 (@) 操作转换到 MLIR 的内置 dialects 部分，并最终生成 LLVM 代码执行。

1.2 实验环境

- 虚拟机软件：VMware Workstation Pro 16.2.3
- 操作系统：Ubuntu 22.04 LTS
- 依赖软件：git version 2.34.1，cmake version 3.22.1，gcc version 11.4.0

2 第一部分

第一部分中我们构建了一个词法分析器来识别 Pony 语言中的各个词法单 (Token)，包括关键字（如 var、def 和 return）、特殊符号、数字以及变量名/函数名等。我们需要通过相关的函数来获取 Token，并对其判断合法性，同时针对非法格式输出响应的报错信息。其中对于成员函数 `getNextChar`，该函数从 curLineBuffer 中获取当前行的下一个 char，如果已经处理到当前行最后一个 char，则通过读取下一行来更新 curLineBuffer 以确保 curLineBuffer 非空。

成员函数 `getTok` 负责实现对于一个 Token 的读取和分析，函数识别标识符的要求如下：

- 能够识别 “return”、“def” 和 “var” 三个关键字；
- 能够识别标识符（函数名，变量名等）：
- 标识符应以字母开头；
- 标识符由字母、数字或下划线组成；
- 按照使用习惯，标识符中不能出现连续的下划线；
- 按照使用习惯，要求标识符中有数字时，数字须位于标识符末尾；

- 在识别每种 Token 的同时，将其存放在某种数据结构中，以便最终在终端输出

此外函数还需要改进识别数字的方法，使编译器可以识别并在终端报告非法数字，非法表示包括：9.9.9，9..9，.999，..9，9.. 等。此处我们定义两种非法输入，一种是数字中含有多个小数点，另一种是数字的小数点位于开头或者末尾。而对于字符识别，函数中定义了多个布尔型变量来记录错误类型，这样便于读入整个错误字符后返回报错并丢弃当前错误字符，输出中将使用 ERROR_TOKEN 来代替该错误字符。主要分为：连续下划线错误、以下划线开头错误、数字不在末尾错误。

最后我们需要完善 dumpToken 函数，作用是使用 lexer 遍历整个文档，最终按顺序输出识别到的每一种 Token。输出格式为：首先输出全部的报错信息，其次输出读取到的全部 token，每个 token 用空格隔开，非法的 token 使用 *"ERROR_TOKEN"* 来代替。上述核心代码在第一部分报告中已详细阐述，此处不再赘述。

3 第二部分

在第二部分，我们需要构建一个语法分析器，将获得的词法单元序列构建成一棵抽象语法分析树 (AST)。具体包括解析函数的声明和调用，Tensor 变量的声明以及 Tensor 的二元运算表达式等，并针对非法格式输出错误信息。首先我们需要完善 parseDeclaration 函数，用于实现对语法变量 “var” 的识别，以及对变量名 “identifier” 以及其对应的初始化形式的 tensor shape 的识别。同时我们要支持以下三种初始化形式：

- var a = [[1, 2, 3], [4, 5, 6]];
- var a<2,3> = [1, 2, 3, 4, 5, 6];
- var<2,3> a = [1, 2, 3, 4, 5, 6];

接着我们需要完善 parseIdentifierExpr 函数来解析标识符语句，其可以是简单的变量名，也可以用于函数调用，形式为 ::=identifier 或者 ::=identifier *'('*expression*')'* 。首先获取语句开头的 identifier，然后判断下一个 token 是否为'('，若是则将其视为是一个函数调用，并读取后面的参数列表；若不是则直接返回 Variable 的 AST。读取参数时将所有参数存入 args 数组中，最后需要先判断函数是否为打印输出函数 print ，并根据参数个数判断是否报错。

最后我们需要补充完整 parseBinOpRHS 函数使用递归的方法去解析一个二元表达式。表达式可能像 *'+'*primary *'−'*primary 或者 (*'+'*primary)∗ 这样的表达式。第一个参数 (exprPrec) 表示当前二元运算符的优先级，第二个参数 (lhs) 表示表达式的左侧表达式。同时我们还需要增加对矩阵乘法 @ 的支持，其优先级和矩阵点乘 * 相同。上述核心代码在第二部分报告中已详细阐述，此处不再赘述。

4 第三部分

4.1 PonyCombine.cpp

由于 Pony 语言的 transpose 函数会对矩阵进行转置操作，而对同一个矩阵连续两次转置操作会得到其本身，因此为了消除冗余代码我们需要补充优化 pass 部分的关键代码。在 PonyCombine.cpp 中我们需要完善 SimplifyRedundantTranspose 类实现冗余的矩阵转置的消除。首先我们需要得到当前的转置操作，其次我们检查待转置矩阵是否已经是一个转置矩阵，若是则消除此冗余转置。核心代码如下：

```
matchAndRewrite(TransposeOp op,
                mlir::PatternRewriter &rewriter) const override {
    mlir::Value.
    mlir::Value transposeInput = op.getOperand();
    TransposeOp InputTransposeOp = transposeInput.getDefiningOp<TransposeOp>();
    if (!InputTransposeOp)
        return failure();

    rewriter.replaceOp(op, InputTransposeOp.getOperand());
    return success();
}
```

4.2 Dialect.cpp

在 MLIR 中，高级语言会从高到低转换成不同抽象层级的中间表示，一般称之为 dialect，生成中间代码，并最终生成底层可执行代码。在本节中我们需要完成的是将 pony dialect 转换为 MLIR 内置的 dialect 这一层中间表示。由于第二部分中新增了矩阵相乘操作，因此我们需要在本节中完善矩阵相乘响应的转换操作。首先在 Dialects.cpp 中完善 `GemmOp::inferShapes` 函数用于推断矩阵相乘之后结果的矩阵形状，用于分配结果的存储空间。结果矩阵行数与左矩阵相同，列数与右矩阵相同，函数核心代码如下：

```
void GemmOp::inferShapes() {
  auto lhsTy = getOperand(0).getType().cast<RankedTensorType>();
  auto rhsTy = getOperand(1).getType().cast<RankedTensorType>();
  auto lhsShape = lhsTy.getShape();
  auto rhsShape = rhsTy.getShape();
  auto elementType = lhsTy.getElementType();

  std::vector<int64_t> resultShape = {lhsShape[0], rhsShape[1]};
  auto resultType = RankedTensorType::get(resultShape, elementType);
  getResult().setType(resultType);
}
```

4.3 LowerToAffineLoops.cpp

在此函数中，高级的循环结构被转为仿射循环，仿射循环是一种具有仿射表达式边界和步长的循环。矩阵相乘是利用循环实现的，因此我们需要将矩阵相乘的 dialect 转换到仿射循环的 dialect。原函数中仿射循环的下节和步长都已确定，我们需要根据矩阵的大小来确定循环的上界。此后循环体中需要取出矩阵行列对应的元素相乘并累加到结果上。需要注意的是，左矩阵的列，右矩阵的行需要以 `MemRefType` 的形式读取，否则可能遇到 assertion failed 的情况。函数核心代码如下：

```
auto M = tensorType.getShape()[0]; // Rows of LHS
auto N = tensorType.getShape()[1]; // Columns of RHS
auto K = (operands[0].getType().cast<MemRefType>()).getShape()[1];
upperBounds[0] = M;
upperBounds[1] = N;
```

```
upperBounds[2] = K;

buildAffineLoopNest(
    rewriter, loc, lowerBounds, upperBounds, steps,
    [&](OpBuilder &nestedBuilder, Location loc, ValueRange ivs) {
      typename pony::GemmOp::Adaptor gemmAdaptor(operands);
      auto i = ivs[0], j = ivs[1], k = ivs[2];
     // Generate loads for the element of 'lhs' and 'rhs' at the inner
      auto lhs = nestedBuilder.create<AffineLoadOp>(loc, gemmAdaptor.getLhs(), ValueRange{i, k});
      auto rhs = nestedBuilder.create<AffineLoadOp>(loc, gemmAdaptor.getRhs(), ValueRange{k, j});
      auto mul = nestedBuilder.create<arith::MulFOp>(loc, lhs, rhs);

      // Generate an accumulator for the result of the multiplication.
      auto current = nestedBuilder.create<AffineLoadOp>(loc, alloc, ValueRange{i, j});
      auto updated = nestedBuilder.create<arith::AddFOp>(loc, current, mul);

      // Store the result of the multiplication.
      nestedBuilder.create<AffineStoreOp>(loc, updated, alloc, ValueRange{i, j});
    });

rewriter.replaceOp(op, alloc);
return success();
```

5 实验结果

5.1 第一部分结果

在对词法分析器构建完毕后，可以通过运行测试用例 test_1 至 test_7 来检查词法分析器的正确性。如图1所示：

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_1.pony -emit=token
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , 2 , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_2.pony -emit=token
Error: Invalid identifier at line 9 column 9 :multiple underline in a row
def multiply_transpose ( a , b ) { return transpose ( a ) transpose ( b ) ; } def main ( ) { var ERROR_TOKEN
[ [ 1 , 2 , 3 ] , [ 4 , 5 , 6 ] ] ; var b 2 , 3 [ 1 , 2 , 3 , 4 , 5 , 6 ] ; var c multiply_transpose ( a , b
) ; print ( c ) ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_3.pony -emit=token
Error: Invalid identifier at line 5 column 10 :digit in the middle of the identifier
def main ( ) { var ERROR_TOKEN [ 2 ] [ 3 ] [ 1 , 2 , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_4.pony -emit=token
Error: Invalid identifier at line 5 column 9 :identifier starts with underline
def main ( ) { var ERROR_TOKEN [ 2 ] [ 3 ] [ 1 , 2 , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_5.pony -emit=token
Error: Invalid number at line 5 column 26 :multiple decimal points
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , ERROR_TOKEN , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_6.pony -emit=token
Error: Invalid number at line 6 column 26 :the decimal point is at the beginning or end of the number
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , ERROR_TOKEN , 3 , 4 , 5 , 6 ] ; }
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_7.pony -emit=token
Error: Invalid number at line 5 column 25 :the decimal point is at the beginning or end of the number
def main ( ) { var a [ 2 ] [ 3 ] [ 1 , ERROR_TOKEN , 3 , 4 , 5 , 6 ] ; }
```

图 1: 词法分析器测试结果

5.2 第二部分结果

对语法分析器构建完毕后，可以通过运行测试用例 test_8 至 test_10 来检查词法分析器的正确性。测试结果如图2所示。从输出结果不难发现，语法分析器正确地得到了输入语句的解析，test 中正确解析了 var<2, 3> a = [1, 2, 3, 4, 5, 6] 的变量声明形式，test_10 中正确解析了矩阵的 @ 运算。

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_8.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_8.pony:4:1
      Params: []
      Block {
        VarDecl a<> @../test/test_8.pony:6:3
          Literal: <2, 3>[ <3>[ 1.000000e+00, 2.000000e+00, 3.000000e+00], <3>[ 4.000000e+00, 5.000000e+00, 6.000000e+00]] @../test/test_8.pony:6:11
        VarDecl b<2, 3> @../test/test_8.pony:7:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @../test/test_8.pony:7:17
        Print [ @../test/test_8.pony:8:3
          var: a @../test/test_8.pony:8:9
        ]
        Print [ @../test/test_8.pony:9:3
          var: b @../test/test_8.pony:9:9
        ]
      } // Block
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_9.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_9.pony:3:1
      Params: []
      Block {
        VarDecl b<2, 3> @../test/test_9.pony:5:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @../test/test_9.pony:5:17
        Print [ @../test/test_9.pony:6:3
          var: b @../test/test_9.pony:6:9
        ]
      } // Block
root@f965a549d8ea:/home/workspace/pony_compiler/build# ../build/bin/pony ../test/test_10.pony -emit=ast
  Module:
    Function
      Proto 'main' @../test/test_10.pony:3:1
      Params: []
      Block {
        VarDecl a<2, 3> @../test/test_10.pony:5:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @../test/test_10.pony:5:17
        VarDecl b<3, 2> @../test/test_10.pony:6:3
          Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @../test/test_10.pony:6:17
        VarDecl c<> @../test/test_10.pony:7:3
          BinOp: @ @../test/test_10.pony:7:15
            var: a @../test/test_10.pony:7:11
            var: b @../test/test_10.pony:7:15
        Print [ @../test/test_10.pony:8:3
          var: c @../test/test_10.pony:8:9
        ]
      } // Block
```

图 2: 语法分析器测试结果

5.3 第三部分

对代码优化完成后，我们通过输出优化前后的 dialect 来比较其优化效果，test11 的前后结果对应图3，test12 的前后结果对应图4，test13 的前后结果对应图5。

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_11.pony -emit=mlir
module {
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = pony.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>
    %2 = pony.reshape(%1 : tensor<6xf64>) to tensor<2x3xf64>
    %3 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %4 = pony.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>
    %5 = pony.reshape(%4 : tensor<6xf64>) to tensor<2x3xf64>
    %6 = pony.add %0, %3 : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %7 = pony.add %2, %5 : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %8 = pony.mul %6, %7 : tensor<*xf64>
    pony.print %8 : tensor<*xf64>
    pony.return
  }
}
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_11.pony -emit=mlir -opt
module {
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = pony.add %0, %0 : tensor<2x3xf64>
    %2 = pony.mul %1, %1 : tensor<2x3xf64>
    pony.print %2 : tensor<2x3xf64>
    pony.return
  }
}
```

图 3: test11 优化前后 dialect

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_12.pony -emit=mlir
module {
  pony.func private @multiply_transpose(%arg0: tensor<*xf64>, %arg1: tensor<*xf64>) -> tensor<*xf64> {
    %0 = pony.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = pony.transpose(%arg1 : tensor<*xf64>) to tensor<*xf64>
    %2 = pony.mul %0, %1 : tensor<*xf64>
    pony.return %2 : tensor<*xf64>
  }
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = pony.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>
    %2 = pony.reshape(%1 : tensor<6xf64>) to tensor<2x3xf64>
    %3 = pony.generic_call @multiply_transpose(%0, %2) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %4 = pony.generic_call @multiply_transpose(%0, %2) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<*xf64>
    %5 = pony.transpose(%0 : tensor<2x3xf64>) to tensor<*xf64>
    %6 = pony.mul %5, %3 : tensor<*xf64>
    %7 = pony.transpose(%2 : tensor<2x3xf64>) to tensor<*xf64>
    %8 = pony.add %6, %7 : tensor<*xf64>
    %9 = pony.add %8, %4 : tensor<*xf64>
    pony.print %9 : tensor<*xf64>
    pony.return
  }
}
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_12.pony -emit=mlir -opt
module {
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = pony.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>
    %2 = pony.mul %1, %1 : tensor<3x2xf64>
    %3 = pony.mul %1, %2 : tensor<3x2xf64>
    %4 = pony.add %3, %1 : tensor<3x2xf64>
    %5 = pony.add %4, %2 : tensor<3x2xf64>
    pony.print %5 : tensor<3x2xf64>
    pony.return
  }
}
```

图 4: test12 优化前后 dialect

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_13.pony -emit=mlir
module {
  pony.func private @transpose_transpose(%arg0: tensor<*xf64>) -> tensor<*xf64> {
    %0 = pony.transpose(%arg0 : tensor<*xf64>) to tensor<*xf64>
    %1 = pony.transpose(%0 : tensor<*xf64>) to tensor<*xf64>
    pony.return %1 : tensor<*xf64>
  }
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    %1 = pony.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>
    %2 = pony.generic_call @transpose_transpose(%1) : (tensor<2x3xf64>) -> tensor<*xf64>
    pony.print %2 : tensor<*xf64>
    pony.return
  }
}
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_13.pony -emit=mlir -opt
module {
  pony.func @main() {
    %0 = pony.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    pony.print %0 : tensor<2x3xf64>
    pony.return
  }
}
```

图 5: test13 优化前后 dialect

test11 到 test13 经过转换得到的内置 dialect 如图6，7所示。

6 总结

总的来说本次实验过程比较顺利，得益于老师和助教在微信群中的悉心答疑以及精心准备的实验指导书。从最开始词法分析器的构建逐渐熟悉整个 Pony 语言与编译器的框架，后续语法分析和代码优化部分的逻辑也十分清晰，主要的困难还是在于对于数据类型的确定，对函数的使用。同时在这个过程中也遇到了一些报错，例如在识别 identifier 时缺少调用 getNext 来获取下一个 token 导致一系列报错，以及在转换内置 dialect 时使用 tensorType 读取参数导致 core dumped 等错误。

此次大作业分阶段布置减轻了不少工作量，也让大家不至于在最后时间去做，在学期中结合课题知识和课后作业对各部分大作业也有更好的理解。此外，希望在前面学习词法分析阶段尚未发布大作业时先让同学们了解基本工具的一些使用例如 docker 等，也可以给一些简单的材料熟悉一下大作业框架，等到学完对应知识再发布正式的大作业。

最后再次感谢赵洁茹老师一学期以来的教学和指导，感谢张炜创和刘广达两位位助教学长本学期在课程作业课后答疑以及大作业相关答疑方面的帮助。

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_11.pony -emit=mlir-affine
module {
  func @main() {
    %cst = arith.constant 6.000000e+00 : f64
    %cst_0 = arith.constant 5.000000e+00 : f64
    %cst_1 = arith.constant 4.000000e+00 : f64
    %cst_2 = arith.constant 3.000000e+00 : f64
    %cst_3 = arith.constant 2.000000e+00 : f64
    %cst_4 = arith.constant 1.000000e+00 : f64
    %0 = memref.alloc() : memref<2x3xf64>
    %1 = memref.alloc() : memref<2x3xf64>
    %2 = memref.alloc() : memref<2x3xf64>
    affine.store %cst_4, %2[0, 0] : memref<2x3xf64>
    affine.store %cst_3, %2[0, 1] : memref<2x3xf64>
    affine.store %cst_2, %2[0, 2] : memref<2x3xf64>
    affine.store %cst_1, %2[1, 0] : memref<2x3xf64>
    affine.store %cst_0, %2[1, 1] : memref<2x3xf64>
    affine.store %cst, %2[1, 2] : memref<2x3xf64>
    affine.for %arg0 = 0 to 2 {
      affine.for %arg1 = 0 to 3 {
        %3 = affine.load %2[%arg0, %arg1] : memref<2x3xf64>
        %4 = affine.load %2[%arg0, %arg1] : memref<2x3xf64>
        %5 = arith.addf %3, %4 : f64
        affine.store %5, %1[%arg0, %arg1] : memref<2x3xf64>
      }
    }
    affine.for %arg0 = 0 to 2 {
      affine.for %arg1 = 0 to 3 {
        %3 = affine.load %1[%arg0, %arg1] : memref<2x3xf64>
        %4 = affine.load %1[%arg0, %arg1] : memref<2x3xf64>
        %5 = arith.mulf %3, %4 : f64
        affine.store %5, %0[%arg0, %arg1] : memref<2x3xf64>
      }
    }
    pony.print %0 : memref<2x3xf64>
    memref.dealloc %2 : memref<2x3xf64>
    memref.dealloc %1 : memref<2x3xf64>
    memref.dealloc %0 : memref<2x3xf64>
    return
  }
}
```

(a) test11 内置 dialect

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_13.pony -emit=mlir-affine
module {
  func @main() {
    %cst = arith.constant 6.000000e+00 : f64
    %cst_0 = arith.constant 5.000000e+00 : f64
    %cst_1 = arith.constant 4.000000e+00 : f64
    %cst_2 = arith.constant 3.000000e+00 : f64
    %cst_3 = arith.constant 2.000000e+00 : f64
    %cst_4 = arith.constant 1.000000e+00 : f64
    %0 = memref.alloc() : memref<2x3xf64>
    affine.store %cst_4, %0[0, 0] : memref<2x3xf64>
    affine.store %cst_3, %0[0, 1] : memref<2x3xf64>
    affine.store %cst_2, %0[0, 2] : memref<2x3xf64>
    affine.store %cst_1, %0[1, 0] : memref<2x3xf64>
    affine.store %cst_0, %0[1, 1] : memref<2x3xf64>
    affine.store %cst, %0[1, 2] : memref<2x3xf64>
    pony.print %0 : memref<2x3xf64>
    memref.dealloc %0 : memref<2x3xf64>
    return
  }
}
```

(b) test13 内置 dialect

图 6: test11 与 test13 测试样例转换内置 dialect 结果

```
root@f965a549d8ea:/home/workspace/pony_compiler/build# ./bin/pony ../test/test_12.pony -emit=mlir-affine
module {
  func @main() {
    %cst = arith.constant 6.000000e+00 : f64
    %cst_0 = arith.constant 5.000000e+00 : f64
    %cst_1 = arith.constant 4.000000e+00 : f64
    %cst_2 = arith.constant 3.000000e+00 : f64
    %cst_3 = arith.constant 2.000000e+00 : f64
    %cst_4 = arith.constant 1.000000e+00 : f64
    %0 = memref.alloc() : memref<3x2xf64>
    %1 = memref.alloc() : memref<3x2xf64>
    %2 = memref.alloc() : memref<3x2xf64>
    %3 = memref.alloc() : memref<3x2xf64>
    %4 = memref.alloc() : memref<3x2xf64>
    %5 = memref.alloc() : memref<2x3xf64>
    affine.store %cst_4, %5[0, 0] : memref<2x3xf64>
    affine.store %cst_3, %5[0, 1] : memref<2x3xf64>
    affine.store %cst_2, %5[0, 2] : memref<2x3xf64>
    affine.store %cst_1, %5[1, 0] : memref<2x3xf64>
    affine.store %cst_0, %5[1, 1] : memref<2x3xf64>
    affine.store %cst, %5[1, 2] : memref<2x3xf64>
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %6 = affine.load %5[%arg1, %arg0] : memref<2x3xf64>
        affine.store %6, %4[%arg0, %arg1] : memref<3x2xf64>
      }
    }
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %6 = affine.load %4[%arg0, %arg1] : memref<3x2xf64>
        %7 = affine.load %4[%arg0, %arg1] : memref<3x2xf64>
        %8 = arith.mulf %6, %7 : f64
        affine.store %8, %3[%arg0, %arg1] : memref<3x2xf64>
      }
    }
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %6 = affine.load %4[%arg0, %arg1] : memref<3x2xf64>
        %7 = affine.load %3[%arg0, %arg1] : memref<3x2xf64>
        %8 = arith.mulf %6, %7 : f64
        affine.store %8, %2[%arg0, %arg1] : memref<3x2xf64>
      }
    }
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %6 = affine.load %2[%arg0, %arg1] : memref<3x2xf64>
        %7 = affine.load %4[%arg0, %arg1] : memref<3x2xf64>
        %8 = arith.addf %6, %7 : f64
        affine.store %8, %1[%arg0, %arg1] : memref<3x2xf64>
      }
    }
    affine.for %arg0 = 0 to 3 {
      affine.for %arg1 = 0 to 2 {
        %6 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>
        %7 = affine.load %3[%arg0, %arg1] : memref<3x2xf64>
        %8 = arith.addf %6, %7 : f64
        affine.store %8, %0[%arg0, %arg1] : memref<3x2xf64>
      }
    }
    pony.print %0 : memref<3x2xf64>
    memref.dealloc %5 : memref<2x3xf64>
    memref.dealloc %4 : memref<3x2xf64>
    memref.dealloc %3 : memref<3x2xf64>
    memref.dealloc %2 : memref<3x2xf64>
    memref.dealloc %1 : memref<3x2xf64>
    memref.dealloc %0 : memref<3x2xf64>
    return
  }
}
```

图 7: test12 测试样例转换内置 dialect 结果