编译原理大作业（三）
代码优化与生成

常烁晨-521021910369

一、内容简介

本部分要进行代码优化。重点考虑如何将冗余的代码进行消除。在 Pony 语言中，有内置的对矩阵转置运算的函数 transpose（）。由于对同一个矩阵做两次转置运算后会得到矩阵本身，因此本实验的目标是在检测到连续调用两次 transpose 函数时，不进行运算，直接返回矩阵本身。

由于在转置运算中，需要遍历整个二维数组，这是通过嵌套循环实现的， 这种运算方式十分影响程序的运行速度。因此，检测到这种类型的冗余代码，并将其进行消除是十分必要的。

本部分实验只需要对上述例子进行处理，以此体会编译器对代码的优化方式。

二、功能实现

需要补充的文件为PonyCombine.cpp

```
  matchAndRewrite(TransposeOp op,
                  mlir::PatternRewriter &rewriter) const
override {
    // TODO: Optimize the scenario: transpose(transpose(x))
-> x
    // Step 1: Get the input of the current transpose.
    // Hint: For op, there is a function: op.getOperand(), it
returns the parameter of a TransposeOp and its type is
mlir::Value.

    Value input = op.getOperand();

    // Step 2: Check whether the input is defined by another
transpose. If not defined, return failure().
    // Hint: For mlir::Value type, there is a function you
may use:
    //       template<typename OpTy> OpTy getDefiningOp ()
const
    //       If this value is the result of an operation of
type OpTy, return the operation that defines it

    auto opt = input.getDefiningOp<TransposeOp>();
    if(!opt)
      return failure();

    // step 3: Otherwise, we have a redundant transpose. Use
the rewriter to remove redundancy.
    // Hint: For mlir::PatternRewriter, there is a function
you may use to remove redundancy:
    //       void replaceOp (mlir::Operation *op,
mlir::ValueRange newValues)
    //       Operations of the second argument will be
replaced by the first argument.

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

本部分大作业所需要实现的代码较少，主要任务就是理解优化框架，根据注释的提示，完善相关内容。

1、首先获取转置的输入。调用 getOperand（）函数。该返回值是命名空间 mlir 中的 Value 类型。

2、检查转置函数的输入是否是另一个转置函数。调用getDifiningOP（）函数，检查返回值。如果输入并不是转置函数，则此处不应优化。

3、如果输入是另一个转置函数，则这两个函数的输出结果为原始的输入。只需调用replaceOp（）函数即可。

三、debug

在完成上面函数的修改后，我尝试运行，但发现了一些奇怪的报错如下。可以看到报错内容与上面所写的函数无关。

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_13.pony -emit=mlir -opt
loc("../test/test_13.pony":12:3): error: unable to infer shape of operation with
out shape inference interface
```

报错的位置是 print 函数，编译器无法识别该函数的接口。由于在上一个语法分析的部分中，我们编写了 print 函数的相关内容，因此考虑此处报错是由于上一部分语法分析时函数编写出现了故障。

重新检查语法分析代码，可以发现在 print 函数返回值的地方出现 bug。

```
if (name == "print")
{
  if (args.size() != 1)
    return parseError<ExprAST>("<single arg>", "as argument to print()");
  return std::make_unique<PrintExprAST>(loc, std::move(args[0]));
}
return std::make_unique<CallExprAST>(loc, std::string(name), std::move(args));
```

在上一次作业的编程中，我遗漏了一行代码，即上图中第五行 return 语句。可以发现在没有 return 语句时，print函数检查不报错后，会按照普通的 call 定义函数的方式进行返回。

此处bug确实由于我的疏漏产生。在上一部分语法分析输出 AST 时可以看到，print 输出的情况如下：

```
        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
        Call 'print' [ @../test/test_8.pony:8:3
          var: a @../test/test_8.pony:8:9
```

最后一行输出显示 Call 'print'。当时我没有发现自己输出结果有问题，好在此时发现了问题，并成功解决。解决该问题后，语法分析部分也可以得到正确的输出/运行结果。

```
parallels@ubuntu-linux-22-04-desktop:~/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
        ]
      } // Block
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_8.pony -emit=jit
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
```

四、运行结果

以下展示代码优化的结果，并且按照实验指导说明书的要求，尝试用其他方式运行编译器。

1、首先是未优化的版本：

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../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], [4.000
000e+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>) -> tenso
r<*xf64>
    pony.print %2 : tensor<*xf64>
    pony.return
  }
}
```

2、输入优化指令后：

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../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], [4.000
000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>
    pony.print %0 : tensor<2x3xf64>
    pony.return
  }
}
```

可以看到优化后的版本直接略去了连续两次转置函数的操作，成功将冗余代码进行优化。

3、输出 AST：

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_13.pony -emit=ast
  Module:
    Function
      Proto 'transpose_transpose' @../test/test_13.pony:5:1
      Params: [x]
      Block {
        Return
          Call 'transpose' [ @../test/test_13.pony:6:10
            Call 'transpose' [ @../test/test_13.pony:6:20
              var: x @../test/test_13.pony:6:30
            ]
          ]
      } // Block
    Function
      Proto 'main' @../test/test_13.pony:9:1
      Params: []
      Block {
        VarDecl a<2, 3> @../test/test_13.pony:10:3
          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_13.pony:10:17
        VarDecl b<> @../test/test_13.pony:11:3
          Call 'transpose_transpose' [ @../test/test_13.pony:11:11
            var: a @../test/test_13.pony:11:31
          ]
        Print [ @../test/test_13.pony:12:3
          var: b @../test/test_13.pony:12:9
        ]
      } // Block
```

4、展示运行结果：

```
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_13.pony -emit=jit
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
parallels@ubuntu-linux-22-04-desktop:~/pony_compiler/build$ ../build/bin/pony ..
/test/test_13.pony -emit=jit -opt
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
```

五、后记

首先要像老师和助教学长致歉，因为我的疏忽，上次语法分析的代码遗漏了一行返回指令，导致语法分析器对 print 函数的异常输出，以及无法调用 jit 指令尝试运行结果。

好在本次实验时由于终端报错，在请教学长后了解了错误出现的可能原因，并最终发现语法分析器中的小故障，在本次作业中完成修改。因此在本次实验提交代码时将修改完成的 Parser.h 文件也一并提交。

完成本次作业后，本学期的大作业就全部完成了。通过一个学期的课程学习，我们了解到了编译器的整体框架，以及各部分的工作原理，同时大作业的项目也难度适中，让我们作为初学者，可以在已有函数的提示下，完成一部分编译器所需要做的工作，通过对一个轻量级的 pony 语言翻译过程，尝试并运用了课上所学的词法分析、语法分析、中间代码生成、代码优化等内容。经过一整个学期的编译原理学习，我对编译这项工作有了初步的接触和了解，并期待今后接触更为高端的编译知识。

感谢赵老师以及张学长一学期的辛苦付出，我多次通过请教助教学长，感谢他百忙之中为我解答问题，非常荣幸能与老师学长一起完成这门课程的学习！