

LLVM Pass Practice Report

钟泓逸 522031910522

1 实验说明

实验代码文件为 `legacyArgCnt.cpp`, `newArgCnt.cpp`, 以及相应的 `CmakeLists.txt`, 放在 `llvm/lib/Transform/ArgCnt/` 文件夹下。

实验测试代码为 `testcase1.c`, 以及相应文件 `testcase1.ll`, 放在 `test` 文件夹下。

2 基于 legacy 实现

2.1 引入所需库文件

- `llvm/ADT/Statistic.h`: 用于统计信息的库
- `llvm/IR/Function.h`: `Function` 抽象类, 提供函数接口
- `llvm/Pass.h`: 定义了 `pass` 的基本接口和方法
- `llvm/Support/raw_ostream.h`: 提供标准输入、标准输出接口
- `llvm/IR/Type.h`: 提供 `Type` 类型

2.2 FunctionPass 实现

具体结构实现如下图。

```
9 namespace{
10     struct legacyArgCnt: public FunctionPass{
11         static char ID;
12         legacyArgCnt():FunctionPass(ID){}
13
14         bool runOnFunction(Function &F) override{
15             int argCnt = 0;
16             int fpArgCnt = 0;
17             for(auto &arg : F.args()){
18                 argCnt++;
19                 if(arg.getType()->isFloatingPointTy()){
20                     fpArgCnt++;
21                 }
22             }
23             errs().write_escaped(F.getName())<<"\t"<<argCnt<<"\t"<<fpArgCnt<<"\n";
24             return false;
25         }
26     };
27 }
28 }
```

图 1: legacyArgCnt 结构

总结构体概述:

- `struct legacyArgCnt: public FunctionPass:`
定义结构体 `legacyArgCnt`，继承自 `FunctionPass`，表示该 `pass` 作用于函数上，且函数之间相互独立。
- `static char ID:` 定义了 `pass` 的唯一标识符 `ID`
- `legacyArgCnt():FunctionPass(ID):` 重载构造函数，用于构造 `legacyArgCnt` 结构体
- `bool runOnFunction(Function &F) override:` 定义 `FunctionPass` 实现的虚方法。

runOnFunction 概述:

- 定义 `int` 变量 `argCnt`、`fpArgCnt`，用于参数和 `float_pointing` 参数的计数
- `for(auto &arg : F.args())` 循环:
用 `Function` 类的 `arg()` 函数获取参数列表，并进行遍历操作，每获取一个参数，`argCnt` 计数增加，然后使用 `getType()` 函数获取当前 `arg` 的类型，并使用 `isFloatingPointTy()` 函数进行类型判断，若为 `FloatingPoint`，则 `fpArgCnt` 计数增加。
- `errs().write_escaped(F.getName())<<"\t"<<argCnt<<"\t"<<fpArgCnt<<"\n";:`
打印函数对应的名称和参数数量。
- `return false:` 表示没有修改函数内容

2.3 pass 注册

代码实现如下:

```
30 char legacyArgCnt::ID = 0;
31 static RegisterPass<legacyArgCnt> X("legacy-arg-cnt", "Argument counter realized by legacy.");
```

图 2: RegisterPass

- `char legacyArgCnt::ID = 0:`
定义 `pass` 的标识符 `ID`，用于在 `pass` 管理系统中注册和识别该 `pass`
- `static RegisterPass<legacyArgCnt> X("legacy-arg-cnt", "Argument counter realized by legacy.):`
使用 `RegisterPass` 类来注册 `pass`，并指定注册类型为 `legacyArgCnt`，并定义命令行选项名称为 `legacy-arg-cnt`，"Argument counter realized by legacy." 定义了运行 `opt` 使用 `help` 选项时打印的帮助信息。

3 基于 new__pass__manager 的实现

3.1 引入头文件

```
1 #include "llvm/ADT/Statistic.h"
2 #include "llvm/Support/raw_ostream.h"
3 #include "llvm/IR/Type.h"
4 #include "llvm/Passes/PassBuilder.h"
5 #include "llvm/Passes/PassPlugin.h"
6 #include "llvm/IR/PassManager.h"
```

图 3: include__path

- `llvm/Passes/PassBuilder.h:` 构建 `pass` 管理器，用于注册和安排 `pass`
- `llvm/Passes/PassPlugin.h:` 定义和注册新的 LLVM Pass 插件，使得自定义 `pass` 能动态加载到 `llvm` 中
- `llvm/IR/PassManager.h:` 提供新式 `pass` 管理器

3.2 Pass 实现

具体结构定义如下。

```
10 namespace{
11     struct newArgCnt: public PassInfoMixin<newArgCnt>{
12         PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM){
13             int argCnt = 0;
14             int fpArgCnt = 0;
15             for(auto &arg : F.args()){
16                 argCnt++;
17                 if(arg.getType()->isFloatingPointTy()){
18                     fpArgCnt++;
19                 }
20             }
21             errs().write_escaped(F.getName())<<"\t"<<argCnt<<"\t"<<fpArgCnt<<"\n";
22             return PreservedAnalyses::all();
23         }
24     };
25 }
```

图 4: newArgCnt 实现

结构概述

- `struct newArgCnt: public PassInfoMixin<newArgCnt>:`
定义结构体 `newArgCnt`，继承自新式 pass 管理器类 `PassInfoMixin`
- `PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM):`
定义实现方法，该方法为每个函数上执行 pass 的实现逻辑。FAM 是函数分析管理，用于获取和管理函数的分析结果。
- `PreservedAnalyses::all():` 表示这个 Pass 没有改变任何分析结果

run 函数概述

- 函数信息的统计和打印与 legacy 方法实现相同

3.3 Pass 的注册与加载

```
27 extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
28 llvmGetPassPluginInfo(){
29     return{
30         LLVM_PLUGIN_API_VERSION, "new-arg-cnt", "v0.1",
31         [](PassBuilder &PB){
32             PB.registerPipelineParsingCallback(
33                 [](StringRef PassName, FunctionPassManager &FPM,
34                 ArrayRef<PassBuilder::PipelineElement>){
35                     if(PassName=="new-arg-cnt"){
36                         FPM.addPass(newArgCnt());
37                         return true;
38                     }
39                     return false;
40                 });
41         });
42     };
43 }
44 }
```

图 5: pass 插件的注册与加载

- `::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK:`
定义了 `PassPluginLibraryInfo`, 用于描述 pass 插件的版本和注册信息, `LLVM_ATTRIBUTE_WEAK` 声明了弱链接, 使得插件可以在链接阶段动态加载。
- `llvmGetPassPluginInfo():` 声明该结构体的初始化函数
- `LLVM_PLUGIN_API_VERSION, "new-arg-cnt", "v0.1":` 定义了插件的名称和版本
- `[(PassBuilder &PB):` 定义 pass 构建器, 用于构建和注册 pass
- `registerPipelineParsingCallback:` 注册回调函数
- `[(StringRef PassName, FunctionPassManager &FPM, ArrayRef<PassBuilder::PipelineElement>) ...`
: 定义回调函数体, 当解析到 pass 名称为 `new-arg-cnt` 时, 向 pass 管理器中增加一个 `newArgCnt` 的 pass 实例, 并且返回 `true` 表示成功注册了 pass

4 测试案例

测试案例中定义两个函数: `test1aaa`, `test2bbb`。具体代码如下:

其中 `test1aaa` 共有 3 个参数, 其中 1 个 `fp` 类型参数; `test2aaa` 共有 3 个参数, 其中 2 个 `fp` 类型参数。

```

1  #include "stdio.h"
2
3  void testaaa(int a, float b, int c){
4      return ;
5  }
6
7  void testbbb(int a, float b, double c){
8      return ;
9  }
10
11 int main(){
12     testaaa(1, 0.1, 2);
13     testbbb(1, 0.23, 2.432);
14     testaaa(1, 0.1, 3);
15     return 0;
16 }

```

图 6: `testcase1.c`

执行指令 `$../build/install/bin/clang -O0 -emit-llvm -S testcase1.c -o testcase1.ll`, 将 `c` 文件转换成 `ll` 文件, 看到转换后的文本如下。

```

7  define dso_local void @testaaa(i32 %a, float %b, i32 %c) #0 {
8  entry:
9      %a.addr = alloca i32, align 4
10     %b.addr = alloca float, align 4
11     %c.addr = alloca i32, align 4
12     store i32 %a, i32* %a.addr, align 4
13     store float %b, float* %b.addr, align 4
14     store i32 %c, i32* %c.addr, align 4
15     ret void
16 }
17
18 ; Function Attrs: noinline nounwind optnone uwtable

```

图 7: `testcase1:testaaa.ll`

```

18 ; Function Attrs: noinline nounwind optnone uwtable
19 define dso_local void @testbbb(i32 %a, float %b, double %c) #0 {
20 entry:
21     %a.addr = alloca i32, align 4
22     %b.addr = alloca float, align 4
23     %c.addr = alloca double, align 8
24     store i32 %a, i32* %a.addr, align 4
25     store float %b, float* %b.addr, align 4
26     store double %c, double* %c.addr, align 8
27     ret void
28 }

```

图 8: testcase1:testbbb.ll

5 实验结果验证

首先，添加 CmakeLists.txt，编译 legacyArgCnt.cpp 和 newArgCnt.cpp 成 libFuncArgCnt 模块。

```

1  add_llvm_library( libFuncArgCnt MODULE BUILDTREE_ONLY
2      legacyArgCnt.cpp
3      newArgCnt.cpp
4
5      DEPENDS
6          intrinsics_gen
7          PLUGIN_TOOL
8          opt
9      )
10

```

图 9: CmakeLists.txt

然后在 Transform 文件夹修改 CmakeLists.txt，添加 add_subdirectory(ArgCnt)。编译文件后得到动态库。

在 test 文件夹下执行指令，分别使用 new-arg-cnt 和 legacy-arg-cnt，获得优化输出信息：

```

● zhy@zhy-virtual-machine:~/Desktop/class/llvm_assignment/llvm-project-llvmorg-11.1.0/test$ ./build/install/bin/opt -load-pass-plugin=../build/lib/LLVMArgCnt.so -passes=new-arg-cnt <testcase1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the '-f' option.

testaaa 3      1
testbbb 3      2
main 0         0
● zhy@zhy-virtual-machine:~/Desktop/class/llvm_assignment/llvm-project-llvmorg-11.1.0/test$ ./build/install/bin/opt -load ../build/lib/LLVMArgCnt.so -legacy-arg-cnt <testcase1.ll
WARNING: You're attempting to print out a bitcode file.
This is inadvisable as it may cause display problems. If
you REALLY want to taste LLVM bitcode first-hand, you
can force output with the '-f' option.

testaaa 3      1
testbbb 3      2
main 0         0

```

图 10: 实验结果

则 pass 处理时正确输出了函数名以及对应参数数量。