编译原理实验报告四

陈永恒 151220012

前言

由于实验三已经把基本的实验内容全部完成了。在老师的建议下，在实验四想研究 `LLVM` 和 `gcc` 的优化。但因为 `gcc` 的规模比较大，模块之间的耦合度很高，学习成本太高且不通用。于是这次选择研究 `LLVM` 是如何优化中间代码的。

LLVM的代码优化

`pass` 是 `LLVM` 中用来优化，分析，修改 `LLVM` 的生成中间代码后，是 `LLVM` 非常重要的组成部分。可以运行在指令，基本块，函数，程序等不同的单元上。
`LLVM` 自带的代码优化 `pass` 有

- always-inline: Inliner for always_inline functions
- argpromotion: Promote 'by reference' arguments to scalars
- constmerge: Merge Duplicate Global Constants
- constprop: Simple constant propagation
- deadargelim: Dead Argument Elimination
- die: Dead Instruction Elimination
- dse: Dead Store Elimination
- instcombine: Combine redundant instructions
- ……

这里只列举了一部分，可以看到几乎包含了各个方面的优化，其中和课程内容相关的有如 `dse` ，消除多余的 `load/store` 指令， `constprop` ，常量替换等等。研究这些代码便可以理解其优化的原理，如果要自定义一个 `pass` 也可以用这些来做基础。所以这次从实验的重点就在分析这些代码。

实验说明

环境搭建

这次由于要编译 `LLVM pass` ，因此我们必须从源代码开始编译 `LLVM` 。
搭建步骤:

1. 下载 `LLVM` 源代码，解压到 `/path/to/source`
2. 新建一个文件夹 `mybuild` ,执行 `cd mybuild; cmake /path/to/source`
3. 最后执行 `cmake --build .`

编译时间大概要1到2个小时，而且编译出来的文件大小超过 `20G` ...简直令人发指。

LLVM Pass

一个 `LLVM function pass` 的结构如下

```
1. #include "llvm/Pass.h"
2. #include "llvm/IR/Function.h"
3. #include "llvm/Support/raw_ostream.h"
4.
5. using namespace llvm;
```

```
6.
7.  namespace {
8.  struct Hello : public FunctionPass {
9.    static char ID;
10.   Hello() : FunctionPass(ID) {}
11.
12.   bool runOnFunction(Function &F) override {
13.     errs() << "Hello: ";
14.     errs().write_escaped(F.getName()) << '\n';
15.     return false;
16.   }
17. }; // end of struct Hello
18. }  // end of anonymous namespace
19.
20. char Hello::ID = 0;
21. static RegisterPass<Hello> X("hello", "Hello World Pass",
22.                               false /* Only looks at CFG */,
23.                               false /* Analysis Pass */);
24.
```

这个 pass 只是单纯的把程序定义的函数名打印出来。对于一个 pass 而言，最主要的是定义 runOnFunction (或 runOnSCC 等)这个函数，这个函数的参数是当前遍历的函数引用。我们可以通过这个参数来访问这个函数的每一条指令和变量。

constprop: Simple constant propagation

这个 pass 的主要功能是实现常量的替换和合并。如指令

```
add i32 1, 2
```

会变成

```
i32 3
```

源代码:

```
1.  bool ConstantPropagation::runOnFunction(Function &F) {
2.    if (skipFunction(F))
3.      return false;
4.
5.    // Initialize the worklist to all of the instructions ready to process...
6.    std::set<Instruction*> WorkList;
7.    for (Instruction &I: instructions(&F))
8.      WorkList.insert(&I);
9.
10.   bool Changed = false;
11.   const DataLayout &DL = F.getParent()->getDataLayout();
12.   TargetLibraryInfo *TLI =
13.       &getAnalysis<TargetLibraryInfoWrapperPass>().getTLI();
14.
15.   while (!WorkList.empty()) {
16.     Instruction *I = *WorkList.begin();
17.     WorkList.erase(WorkList.begin());    // Get an element from the worklist...
18.
19.     if (!I->use_empty())                 // Don't muck with dead instructions...
20.       if (Constant *C = ConstantFoldInstruction(I, DL, TLI)) {
21.         // Add all of the users of this instruction to the worklist, they might
22.         // be constant propagatable now...
23.         for (User *U : I->users())
24.           WorkList.insert(cast<Instruction>(U));
25.
26.         // Replace all of the uses of a variable with uses of the constant.
27.         I->replaceAllUsesWith(C);
28.
29.         // Remove the dead instruction.
30.         WorkList.erase(I);
```

```
        if (isInstructionTriviallyDead(I, TLI)) {
          I->eraseFromParent();
          ++NumInstKilled;
        }

        // We made a change to the function...
        Changed = true;
      }
  }
  return Changed;
}
```

流程如下:

1. 收集函数的每一条指令。
2. 对于每一条指令，调用 `ConstantFoldInstruction` 函数看是否能够将指令变成常量。
3. 如果可以，找到所有的使用了这条指令(的结果)的指令，并把它替换成常量

查看 `ConstantFoldInstruction` 的源代码可以发现，判断一条指令是否能被折叠，简单的我们可以直接把操作数扫描一遍，如果所有的操作数都是常量，那么可以直接用常量替换即可，其中一些如 `const int i=3` 这种也能被替换。

argpromotion: Promote 'by reference' arguments to scalars

这个 `pass` 试图去用值传递来替换引用传递。
分析源代码可以发现(源代码有1k多行，就不贴出来了orz..)，这个的 `pass` 的机制大致是观察传入的参数中为引用参数的部分(如指针传递)。如果对于这个指针的使用仅限于 `load` ，而没有 `store` ，那么我们就可以把引用传递替换成值传递。这样组的好处是我们可以递归的简化代码，达到消除局部变量的分配和引用，特别是对于像 `STL` 这种模板类优化效果会比较好。

对于这个 `pass` ，我做了以下的实验

```
#include <stdio.h>

int hello(int* p){
    int k=*p;
    return k+1;
}
int main(){

    int i=1;
    hello(&i);
    return 0;
}
```

编译: `clang -m32 -O3 -emit-llvm hello.c -c -o hello.bc`
反编译到汇编: `llvm-dis < hello.bc`
他生成的中间代码为

```
; Function Attrs: noinline nounwind optnone
define i32 @hello(i32*) #0 {
  %2 = alloca i32*, align 4
  %3 = alloca i32, align 4
  store i32* %0, i32** %2, align 4
  %4 = load i32*, i32** %2, align 4
  %5 = load i32, i32* %4, align 4
  store i32 %5, i32* %3, align 4
  %6 = load i32, i32* %3, align 4
  %7 = add nsw i32 %6, 1
  ret i32 %7
}

; Function Attrs: noinline nounwind optnone
```

```
15. define i32 @main() #0 {
16. //.......
17. }
```

发现并没有优化。函数中参数的定义依然是一个指针。如果按照文档的说明生成的中间代码应该是类似于

```
1. define i32 @hello(i32*) #0 {
2.   %2 = alloca i32*, align 4
3.   %5 = load i32, i32* %0, align 4
4.   store i32 %5, i32* %3, align 4
5.   %6 = load i32, i32* %3, align 4
6.   %7 = add nsw i32 %6, 1
7.   ret i32 %7
8. }
```

可能是我这个程序太简单了。但这个 pass 的思想还是挺好的，因为这样的就可以减少 load 这种访存操作了。

deadargelim: Dead Argument Elimination

这个 pass 主要是消除无用的参数。分析源代码可以发现它的机制如下

1. 对于参数数量不确定的函数的调用，我们看看是否能将它的参数个数确定下来，这样可以减少 va_start 这种函数的调用次数。
2. 扫描函数指令，用数据流分析的方法确定一个函数参数是否 dead ，如在被使用前就定值。如果是，我们在传参的时候可以忽略这个参数。
3. 查看是否有没有使用的参数，如果有，可以直接将其忽略。

像这个例子

```
1. #include <stdio.h>
2.
3. int hello(int i,int* p){
4.     int k=i;
5.     return k+1;
6. }
7. int main(){
8.
9.     int i=1;
10.    hello(i,&i);
11.    return 0;
12. }
```

如果使用了 -deadargelim ，编译出来的IR是下面这样的

```
1. define i32 @hello(i32) #0 {
2.   %3 = alloca i32, align 4
3.   %4 = alloca i32*, align 4
4.   %5 = alloca i32, align 4
5.   store i32 %0, i32* %3, align 4
6.   store i32* %1, i32** %4, align 4
7.   %6 = load i32, i32* %3, align 4
8.   store i32 %6, i32* %5, align 4
9.   %7 = load i32, i32* %5, align 4
10.  %8 = add nsw i32 %7, 1
11.  ret i32 %8
12. }
```

可以看到hello的参数变成了一个，因为第二个参数根本没有被使用，因此被优化掉了。
我们看看下面将 clang 的优化选项开到 O3 的代码是怎么样的:

```
define i32 @hello(i32, i32* nocapture readnone) local_unnamed_addr #0 {
  %3 = add nsw i32 %0, 1
  ret i32 %3
}
```

只有两句汇编。可以看到代码极大的被优化，首先是把局部变量优化了，应该是要直接使用寄存器。然后第二个参数还在，但是被标记成了 `nocapture readnone` 。只能说有点厉害。。

其他

还有一些比较厉害的 pass
`instcombine` :合并冗余指令

```
%Y = add i32 %X, 1
%Z = add i32 %Y, 1
```

会变成

```
%Z = add i32 %X, 2
```

`indvars` : 变量归纳和强度消减

```
for (i = 7; i*i < 1000; ++i)
```

会变成

```
for (i = 0; i != 25; ++i)
```

总结

这次的实验我研究了 `llvm` 的代码优化方式。可以看到， `llvm` 把代码优化分成了一个个 `pass` ,每一个 `pass` 可以只专注于做一种优化。这样使得代码优化变得相对简单。唯一的不足是这样会线性增长优化时间，因为每一个 `pass` 至少会遍历一次代码，而要很好的优化 `llvm` 的中间代码，必须同时使用至少10个以上的 `pass` ，而且每个 `pass` 可能不止遍历一次代码。但我觉得这个是可以接受的。因为考虑到程序使用多次的话，这部分的成本是可以被均摊的。

整个实验我采用了 `llvm` 来实现。不得不说 `llvm` 真的很强大，实现了龙书中 `m` 个前端， `n` 个后端就可以支持 `m*n` 种语言的目标。而且 `llvm` 的模块化做得很出色，可以让开发人员专注于一项任务。在学习了编译原理课程后去研读它的源代码也比较轻松。可以说这次的实验让我受益匪浅。