编译原理研讨课实验PR001实验报告

第十二组 曹翔舟 万之凡 钟赟

任务说明

课堂任务：熟悉实验环境操作

1. 掌握如何从源代码编译安装LLVM和Clang
2. 了解如何生成和查看C程序对应的AST

   整体操作较为简单且被任务二涵盖，因此不做赘述

添加element wise操作的制导实验内容：

在程序中添加element wise操作的指导，区域以函数/过程的定义(不是声明)为单位，形如：

```
#pragma elementWise
int func_name(){
    .....
}
```

对于添加了制导的源程序`*.c`，按照规则在编译时打印每个函数的名称，该函数是否在制导范围内。

对于一个函数的是否在制导范围内的定义：

`1. 一个制导总是匹配在其后出现的，离它最近的一个函数定义`

`2. 一个制导只能匹配最多一个函数定义`

实验具体流程

在具体的实验流程中走了很多弯路，特此汇总。

LLVM与Clang的安装

建立构建目录(Out of Source)：`mkdir -p build && cd build`

通过CMake生成GNU标准的Makefile，并使用make进行编译和安装:

1. 设定编译过程使用的*gcc*和*g++*: `export CC=/usr/local/bin/gcc && export CXX=/usr/local/bin/g++`
2. 生成Makefile：`cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=~/llvm ../llvm-3.3`，其中*CMAKE_INSTALL_PREFIX*代表了编译完成以后的安装目录，*../llvm-3.3*代表了源代码目录。

   注意这一步的目录已经切换到了`/clang12/build`下
3. 使用*make*进行编译：`make -j6`，其中*j6*代表使用6个线程并行编译。该过程约需要10分钟。如果仅仅是更改了*Clang*的源代码，则无需执行第二步。

   在服务器空闲时可以适当增加编译线程数提速，"无需执行第二步"在后期修改输出内容时体现
4. 安装到`CMAKE_INSTALL_PREFIX`当中：`make install`
5. 检查`~/llvm`下是否已经有对应的文件安装进去：`~/llvm/bin/clang --version`，将会有`clang version 3.3 (tags/RELEASE_33/final)`对应字样输出

代码完成后编译

第一步，从clang0用户下拷贝代码：`scp -r clang0@124.16.71.6:/home/clang0/src/examples ~/llvm-3.3/tools/clang/`（请注意根据自己的clang目录调整目标目录，这个是以教程为例的说明）

第二步，编译LLVM/Clang：

1.生成Makefile：`cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=~/llvm ../llvm-3.3`（生成位置和第一步一致）

2.使用*make*进行编译：`make -j6`

3.安装到`CMAKE_INSTALL_PREFIX`当中：`make install`

第三步，编译安装Plugin。`cd ~/build/tools/clang/examples/TraverseFunctionDecls && make`。此时，可以通过如下命令验证是否成果：`file ~/build/lib/TraverseFunctionDecls.so`，如果相应的文件存在，则说明成功。

不存在很有可能是第一步copy时产生的问题

第四步，执行。`~/llvm/binclang -cc1 -load ~/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/example.c`。请注意，如果对于不合法的pragma，同学们的实现是给出警告，而不是错误，那么请在作业的`compile_and_check.sh`的编译选项中加入`-Werror`，形如`~/llvm/binclang -cc1 -load ~/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/example.c -Werror`，可以利用`echo $?`来判断是否符合预期。

代码运行

最后使用写好的脚本和测试集进行测试：`./compile_and_check.sh /ruote/test.c`

成员组成

姓名：万之凡 学号：2016K8229929049

姓名：钟 赟 学号：2016K8009915009

姓名：曹翔舟 学号：2016K8009929027

实验设计

设计思路

本次实验需要Preprocessor/Lexer/Parser/Sema的协同工作。

1. Lexer阶段

- 编译器读到`#pragma`时，调用`HandlePragma`函数。
- `HandlePragma`函数调用词法分析器获得下一个Token，直到读到制导语句结束标识符`eod`。在这个过程中，如果`#pragma elementWise`与`eod`之间存在其他记号，则报warning。
- 声明一个`annot_pragma_elementwise`类型的Token，并将该Token插入符号流。
- 根据Token，寻找合适的Handle。

2. Paser阶段

- 如果读到的Token为`elementWise`，则调用`HandlePragmaElementWise`函数；紧接着根据其中eof下一个记号的位置ahd来执行`ActOnPragmaElementWise`函数，得到函数定义在文件中的偏移量并存入。
- 上述的ahd位置和Token位置对应的文件ID可以解决调用等问题。

3. Sema阶段

- 之后读到一个函数定义 ND时会调用`getAsCheckRule`函数（通过自己实现），用于判断是否需要将其做AsCheck，即判断是否符合`#pragma elementWise`，直接输出结果即可。

实验实现

详细代码见下：

- `llvm-3.3/tools/clang/lib/Parse/Parser.cpp`

```
ElementWiseHandler.reset(new PragmaElementWiseHandler());
PP.AddPragmaHandler(ElementWiseHandler.get());
```

```
PP.RemovePragmaHandler(ElementWiseHandler.get());
ElementWiseHandler.reset();
```

```
  case tok::annot_pragma_elementWise:
      HandlePragmaElementWise();
      return DeclGroupPtrTy();
```

- `llvm-3.3/tools/clang/lib/Parse/ParsePragma.cpp`

```
void Parser::HandlePragmaElementWise(){
    assert(Tok.is(tok::annot_pragma_elementWise));
    SourceLocation PragmaLoc = ConsumeToken();
    Actions.ActOnPragmaElementWise(PragmaLoc);
}
```

```
void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
                        PragmaIntroducerKind Introducer,
                        Token &EWTok) {
    SourceLocation EWLoc = EWTok.getLocation();
    PP.CheckEndOfDirective("pragma elementWise");

    Token *Toks =
        (Token*) PP.getPreprocessorAllocator().Allocate(sizeof(Token) * 1,
         llvm::alignOf<Token>());
    new (Toks) Token();
    Toks[0].startToken();
    Toks[0].setKind(tok::annot_pragma_elementWise);
    Toks[0].setLocation(EWLoc);
    Toks[0].setAnnotationValue(0);
    PP.EnterTokenStream(Toks, 1, /*DisableMacroExpansion=*/true,
/*OwnsTokens=*/false);
}
```

- `llvm-3.3/tools/clang/lib/Parse/ParseStmt.cpp`

```
case tok::annot_pragma_elementWise:
    ProhibitAttributes(Attrs);
    HandlePragmaElementWise();
    return StmtEmpty();
```

```
case tok::annot_pragma_elementWise:
    HandlePragmaElementWise();
    break;
```

- `llvm-3.3/tools/clang/lib/Parse/ParseDeclCXX.cpp`

```
if(Tok.is(tok::annot_pragma_elementWise)){
    HandlePragmaElementWise();
    continue;
}
```

- `llvm-3.3/tools/clang/lib/Parse/ParseDecl.cpp`

```
if(Tok.is(tok::annot_pragma_elementWise)){
    HandlePragmaElementWise();
    continue;
}
```

- `llvm-3.3/tools/clang/lib/Sema/Sema.cpp`

```
/*change*/ElementWisePragmaOn(false), /*end change*/
```

```
ElementWisePragmaOn = false;
```

- `llvm-3.3/tools/clang/lib/Sema/SemaDecl.cpp`

```
if(ElementWisePragmaOn){
    FD->setElementWise(true);
    ElementWisePragmaOn = false;
}
```

```
void Sema::ActOnPragmaElementWise(SourceLocation PragmaLoc){
    ElementWisePragmaOn = true;
}
```

- `llvm-3.3/tools/clang/include/clang/Parse/Parser.h`

```
OwningPtr<PragmaHandler> ElementWiseHandler;
```

```
/// \brief Handle the annotation token produced for
/// #pragma elementWise...
void HandlePragmaElementWise();
```

- `llvm-3.3/tools/clang/include/clang/Sema/Sema.h`

```
bool ElementWisePragmaOn;
```

```
void ActOnPragmaElementWise(SourceLocation PragmaLoc);
```

- `llvm-3.3/tools/clang/include/clang/AST/Decl.h`

```
bool IsElementWise : 1;
```

```
IsElementWise(false),
```

```
bool getElementWise() const { return IsElementWise; }
void setElementWise(bool E = true){IsElementWise = E;}
```

可以看到大部分添加的内容不多，集中的改动主要是添加`#pragma elementWise`类型的新判断标签，很多bool类型和case类型的函数都是和上下文相似，可能展示不全，需要在具体代码中结合上下文去分析判断。

其它

- `llvm-3.3/tools/clang/include/clang/Basic/DiagnosticLexKinds.td`

```
def pp_pragma_element_wise_found : Warning<"ElementWise found">;
```

- `llvm-3.3/tools/clang/include/clang/Basic/DiagnosticParseKinds.td`

```
def warn_pragma_elementWise_found : Warning<
    "pragma elementWise found in front of this function definition">;
```

- `llvm-3.3/tools/clang/include/clang/Basic/TokenKinds.def`

```
// Annotation for #pragma elementWise...
// The lexer produces these so that they only take effect when the parser
// handles them.
ANNOTATION(pragma_elementWise)
```

这是区别于判断逻辑的部分，在tokens中创建新的类型。

总结

实验结果总结

运行命令：

```
./compile_and_check.sh $PATH        // $PATH为测试文件路径
```

P.S. 本次实验的测试文件位于`~/PROO1/scripts/test`文件夹中。

测试文件如下：

```
void f();

#pragma elementWise
void f1(){}

#pragma elementWise 1
int f2(){}
#pragma elementWise

void f3();
void f4(){}

#pragma elementWise
```

测试结果如下：

```
[clang12@host2 scripts]$ ./compile_and_check.sh test/test.c
'arning: unknown -Werror warning specifier: '-Werror
top-level-decl: "f": elementWise: 0
test/test.c:6:21: warning: extra tokens at end of #pragma elementWise directive
#pragma elementWise 1
                    ^
                    //
top-level-decl: "f1": elementWise: 1
test/test.c:7:10: warning: control reaches end of non-void function
int f2(){}
         ^
top-level-decl: "f2": elementWise: 1
top-level-decl: "f3": elementWise: 0
top-level-decl: "f4": elementWise: 1
3 warnings generated.
[clang12@host2 scripts]$
```

结果显示：

函数`f`为平凡测试，正确输出`elementwise: 0`；

函数`f2`前的制导语句多输入的`1`报出Warning；

函数`f3`是函数声明，故制导语句对其不起作用，正确输出`elementwise: 0`；

函数`f1`、`f2`、`f4`正确输出`elementwise: 1`；

最后的`#pragma elementWise`也多余输出。

实验成功。

分成员总结

曹翔舟

> 本次实验时，由于对于实验需求的理论知识掌握不是很透彻，导致实验进行时对于代码的修改无从下手，没有头绪，在同学的指导下有了一定思路，并且知道了可以定位自己需要修改的函数的简单方法。在实现功能时，对于函数的理解和构建可以参考原有的代码来实现，在以后的学习中可以加以利用。添加的一些操作功能和已知的一些功能流程比较相似，可以进行一定的类比参考。纯文本界面对以前即使在linux中也习惯图形界面的我是个需要经历熟悉的环境，感谢万之凡同学对于服务器命令操作的指导，以及认识到了学会随手百度谷歌可以省去自己和其他同学的不少时间

万之凡

> 实验中大部分的困难来自于c++的语言问题和服务器端的操作问题。c++相较于我们熟识的c、verilog更加灵活方便，但是还是有很多习惯一时间改正不过来，写出了一些语法错误并不明显的辣鸡代码（当然还有顺手打错的沙雕问题），但是还是一点点解决了。另一个大问题就是服务器端的操作，尽管以前做过很多linux系统下的实验，但我个人还是偏好图形界面，在纯命令操作的服务器中走了很多弯路来熟悉实验过程；vim的操作也不是很熟练，尽管全键盘操作很帅但是自己总会顺手ctrl+s锁了界面...自己属于要把实验彻底想清楚才会写的顺利的类型，导致在期间经常叨扰助教，感谢不厌其烦的解答！同样在此感谢雷正宇、侯承轩同学在实验过程中给予的帮助和指点！

钟赟

> 本次实验的难点在于不熟悉实验框架和C++语言，之后老师给出Pragma实现说明文档和三组同学的报告给了很大帮助。在服务器上的编译安装也遇到过很大问题，感谢助教都一一帮忙解决了。后来实现的过程中也有和老师的文档不尽相同的地方，比如最后的测试部分（可能是打开方式不对），改成修改PrintNames.cpp文件。对于我个人，需要加强学习C++语言。最后，感谢助教的指导，感谢队友的鼎力相助！