编译原理研讨课实验PR001实验报告

任务说明

本实验PR001包含两个任务

1. 熟悉Clang的安装和使用

- 正确编译生成Clang和LLVM的可执行文件
- 可以查看一个C程序对应的AST

2. 添加C语言对`#pragma elementWise`操作的支持

- 对于添加了制导的源程序`*.c`，借助提供的插件，在编译时打印每个函数的名称，正确判断该函数是否在制导范围内。
- 判断规则：
  - 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。
  - 一个制导只能匹配最多一个函数定义。

成员组成

骆嵩涛 2018K8009907001
毛啸腾 2018K8009907033
陈彦帆 2018K8009918002

实验设计

设计思路

具体目标

Parser生成AST时，每个函数（定义或声明）会生成一个FunctionDecl结点，在提供的测试方法`TraverseFunctionDeclsVisitor`中，测试规则是检验每个FunctionDecl结点的`getAsCheckRule()`返回值。于是我们的目标是在FunctionDecl结点上增加一个标记属性(IsElementWise)，标记该声明或函数是否被标记上`#pragma elementWise`。`getAsCheckRule()`返回该属性作为判断结果。

实现思路

首先确定标记FunctionDecl的任务无法仅通过Preprocessor完成，应当在Parse阶段完成，因为在Parse阶段生成AST。

在C语言的ParseAST过程中，一个pragma token的上下文可能有以下三种情况:

- 在函数或结构体定义外部
- 在函数内部
- 在结构体定义内部

根据补充说明，只需支持上述第一种情况下的`#pragma elementWise`，其它情况忽略，在我们的实现中，会抛出警告。

首先在Token类为#pragma elementWise添加一种TokenKind: `annot_pragma_elementWise`.

在Sema中添加一个属性(ElementWisePragmaOn)，指示elementWise的作用域，初始化为false。若Parser在顶层遇到`annot_pragma_elementWise` Token，将该属性设为true(通过HandlePragmaElementWise方法)。

将FunctionDecl的elementWise标记(IsElementWise)初始化为false。当Parser进入一个函数定义时，若当前elementWise作用域有效(ElementWisePragmaOn==true)，将该函数的elementWise标记(IsElementWise)设为true，并把作用域设为无效。

这样，测试方法就能正确打印每个函数的IsElementWise属性了。由此，实现了一个elementWise匹配至多一个函数定义。由于clang在Parse连续多个函数时是顺序的，故满足elementWise只匹配最近的函数定义。

实验实现

1. 添加TokenKind::annot_pragma_elementWise

仿照pragma pack:
`./tools/clang/include/clang/Basic/TokenKinds.def`

```
// elementWise
ANNOTATION(pragma_elementWise)
```

2. 添加`PragmaHandler`的子类`PragmaElementWiseHandler`，重写其`HandlePragma`方法，让Preprocessor能正确识别`annot_pragma_elementWise`

仿照PragmaMSStructHandler:
`./tools/clang/lib/Parse/ParsePragma.cpp:PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,`
`PragmaIntroducerKind Introducer,`
`Token &elementWiseTok)`

```
void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
                                            PragmaIntroducerKind Introducer,
                                            Token &elementWiseTok) {
  Token Tok;
  PP.Lex(Tok);
  //判断elementWise后只有换行标号，否则忽略并警告
  if (Tok.isNot(tok::eod)) {
    PP.Diag(Tok.getLocation(), diag::warn_pragma_extra_tokens_at_eol)
      << "elementWise";
    return;
  }
  //创建一个新的Token对象，标上基本属性，并标明TokenKind是annot_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(elementWiseTok.getLocation());
  Toks[0].setAnnotationValue(NULL);
  PP.EnterTokenStream(Toks, 1, /*DisableMacroExpansion=*/true,
                      /*OwnsTokens=*/false);
}
```

定义子类的代码省略。
之后，还需在Parser的构造函数将其注册，在Parser析构函数将其删除。仿照PragmaMSStructHandler即可，代码省略。

3. 在Parse过程中根据Preprocessor返回的TokenKind的值调用`HandlePragmaElementWise`

仿照pragma pack:

- Parse外部声明或定义
  `./tools/clang/lib/Parse/Parser.cpp:Parser::ParseExternalDeclaration(ParsedAttributesWithRange &attrs,`
  `ParsingDeclSpec *DS)`

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

- Parse函数内部的语句或声明
  `./tools/clang/lib/Parse/Parser.cpp:Parser::ParseStatementOrDeclarationAfterAttr`

`ibutes(StmtVector &Stmts,`
`bool OnlyStatement, SourceLocation *TrailingElseLoc,`
`ParsedAttributesWithRange &Attrs)`

```
case tok::annot_pragma_elementWise:
    ProhibitAttributes(Attrs);
    Diag(Tok.getLocation(), diag::warn_pragma_elementWise_scope);
    ConsumeToken();
    //HandlePragmaElementWise(); we ignore
    return StmtEmpty();
```

- Parse结构体定义
  `./tools/clang/lib/Parse/ParseDecl.cpp:Parser::ParseStructUnionBody(SourceLocation RecordLoc,`
  `unsigned TagType, Decl *TagDecl)`

```
if (Tok.is(tok::annot_pragma_elementWise)) {
      Diag(Tok.getLocation(), diag::warn_pragma_elementWise_scope);
      ConsumeToken();
      //HandlePragmaElementWise(); we ignore
      continue;
    }
```

4. 实现HandlePragmaElementWise:开启elementWise作用域(ElementWisePragmaOn)

处理逻辑很简单，消费当前的Token，然后调用当前Parser的Sema类对象Actions的方法`ActOnPragmaElementWise()`，后者将Actions的ElementWisePragmaOn属性设为true，从而开启elementWise作用域。
`./tools/clang/lib/Parse/ParsePragma.cpp`

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

`./tools/clang/lib/Sema/SemaAttr.cpp`

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

ElementWisePragmaOn需要在Sema构造函数初始化为false。(仿照MSStructPragmaOn)
`./tools/clang/lib/Sema/Sema.cpp:Sema::Sema(Preprocessor &pp, ASTContext &ctxt, ASTConsumer &consumer,`
`TranslationUnitKind TUKind,`
`CodeCompleteConsumer *CodeCompleter)`

```
...
PackContext(0), ElementWisePragmaOn(false), MSStructPragmaOn(false),
VisContext(0),
...
```

5. Parser在函数定义开始时将FunctionDecl做上标记，并关闭elementWise作用域。

根据gdb的调试，我们发现Parser在确定一个Decl为函数定义时会进入`ActOnStartOfFunctionDef()`方法。在该方法中给函数对应的FunctionDecl打上标记，并关闭elementWise作用域即可。(该标记在FunctionDecl构造函数初始化为0)

`./tools/clang/lib/Sema/SemaDecl.cpp:Sema::ActOnStartOfFunctionDef(Scope *FnBodyScope, Decl *D)`

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

6. 补充上述实现中省略的声明或定义

`./tools/clang/include/clang/AST/Decl.h`

```
  bool IsElementWise : 1;
...
/// Whether this function is marked as elementWise.
  bool isElementWise() const { return IsElementWise; }
  void setElementWise(bool V) { IsElementWise = V; }
```

在这一部分内容定义了一个FuncDecl的IsElementWise属性，同时声明并定义了有关IsElementWise的两个方法`setElementWise`和`isElementWise`，分别用于将IsElementWise置位和返回IsElementWise的布尔值。

`./tools/clang/include/clang/Sema/Sema.h`

```
  bool ElementWisePragmaOn; // True when \#pragma elementWise on
 ...
  /// ActOnPragmaElementWise - Called on well formed \#pragma elementWise.
  void ActOnPragmaElementWise();
```

在这一部分定义了ElementWisePragmaOn这一全局flag，标记elementWise的作用域。声明了ActOnPragmaElementWise方法。

`./tools/clang/include/clang/Parse/Parser.h`

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

`./tools/clang/lib/Parse/ParsePragma.h`

```
  class PragmaElementWiseHandler : public PragmaHandler {
public:
  explicit PragmaElementWiseHandler() : PragmaHandler("elementWise") {}

  virtual void HandlePragma(Preprocessor &PP, PragmaIntroducerKind Introducer,
                            Token &FirstToken);
};
```

`./tools/clang/lib/Parse/Parser.cpp`

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

定义了PragmaHandler的子类PragmaElementWiseHandler，重写了它的HandlePragma方法。
将ElementWiseHandler的Ptr值初始化为PragmaElementWiseHandler。
注册之后，在调用`FindHandler`来寻找elementWise的handler时能够返回`PragmaElementWiseHandler`的地址。在析构Parser时删除PragmaElementWiseHandler并将ElementWiseHandler的Ptr值设为0。

7. 修改测试方法`TraverseFunctionDeclsVisitor`

`funcNamesToAsCheckRule`是一个存储FunctionDecl信息及其测试结果的map。若有多个同名声明或定义，只会记录其中一个。故记录时需要进行额外的检测以防非零检测值被覆盖。
`./tools/clang/examples/TraverseFunctionDecls/TraverseFunctionDecls.cpp`

```
unsigned rule = FD -> isElementWise();
            if(rule != 0) {
                funcNamesToAsCheckRule[FD->getNameAsString()] = rule;
            } else {
                std::map<std::string, unsigned>::iterator it =
funcNamesToAsCheckRule.find(name);
                if(it == funcNamesToAsCheckRule.end())
                    funcNamesToAsCheckRule[FD->getNameAsString()] = rule;
            }
```

其它

本次代码主要涉及的调用流程

在`./tools/clang/tools/driver/driver.cpp`中进入main函数，本次实验着眼于前端cc1，此时main执行
`return cc1_main(argv.data()+2, argv.data()+argv.size(), argv[0],(void*) (intptr_t) GetExecutablePath);`
`cc1_main`创建了一个CompilerInstance对象Clang, 调用
`Success = ExecuteCompilerInvocation(Clang.get());` 来执行一个编译流程。

`ExecuteCompilerInvocation`调用`OwningPtr<FrontendAction> Act(CreateFrontendAction(*Clang));`后，Act实际指向的对象是
`case ParseSyntaxOnly:  return new SyntaxOnlyAction();`
后者是ASTFrontendAction的一个子类。

随后执行
`bool Success = Clang->ExecuteAction(*Act);`
接着进入`Act.Execute();`和`ExecuteAction()`，然后进入`ParseAST`方法。
在`ParseAST`中

```
    do {
      if (ADecl && !Consumer->HandleTopLevelDecl(ADecl.get()))
        return;
    } while (!P.ParseTopLevelDecl(ADecl));
```

不断调用P.ParseTopLevelDecl方法来解析生成树顶层的声明符，后者最终调用`ParseExternalDeclaration()`来解析顶层的定义或声明。解析过程中，会调用Preprocessor来处理词法，调用Sema来处理语义。

要进入上文实验实现涉及的`ActOnStartOfFunctionDef()`方法，接着要经过`ParseDeclarationOrFunctionDefinition()-->ParseDeclOrFunctionDefInternal()-->ParseDeclGroup()-->ParseFunctionDefinition()-->ActOnStartOfFunctionDef()`

总结

实验结果总结

任务1 熟悉Clang的安装和使用

操作方法在老师提供的实验说明中有详细叙述，这里不再重复。
下面展示了当前目录下test1.c文件的AST生成结果。

```
TranslationUnitDecl 0x6b23e70 <<invalid sloc>>
|-TypedefDecl 0x6b24350 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x6b243b0 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x6b24700 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl 0x6b247a0 <test1.c:2:1, col:17> joo 'char ()' extern
|-FunctionDecl 0x6b24910 <line:3:1, line:9:1> f 'int (int)'
| |-ParmVarDecl 0x6b24850 <line:3:7, col:11> x 'int'
| `-CompoundStmt 0x6b50eb0 <line:4:1, line:9:1>
|   |-DeclStmt 0x6b24a48 <line:5:5, col:14>
|   | `-VarDecl 0x6b249d0 <col:5, col:13> y 'int'
|   |   `-IntegerLiteral 0x6b24a28 <col:13> 'int' 2
|   |-DeclStmt 0x6b24b70 <line:6:5, col:26>
|   | `-VarDecl 0x6b24a70 <col:5, col:25> result 'int'
|   |   `-ParenExpr 0x6b24b50 <col:18, col:25> 'int'
|   |     `-BinaryOperator 0x6b24b28 <col:19, col:23> 'int' '/'
|   |       |-ImplicitCastExpr 0x6b24b10 <col:19> 'int' <LValueToRValue>
|   |       | `-DeclRefExpr 0x6b24ac8 <col:19> 'int' lvalue ParmVar 0x6b24850
'x' 'int'
|   |       `-IntegerLiteral 0x6b24af0 <col:23> 'int' 42
|   `-ReturnStmt 0x6b50e90 <line:8:5, col:12>
|     `-ImplicitCastExpr 0x6b50e78 <col:12> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x6b50e50 <col:12> 'int' lvalue Var 0x6b24a70 'result'
'int'
|-RecordDecl 0x6b50ee0 <line:10:1, line:15:1> struct g
| |-FieldDecl 0x6b50fb0 <line:13:5, col:9> x 'int'
| `-FieldDecl 0x6b51010 <line:14:5, col:10> aa 'long'
|-FunctionDecl 0x6b510b0 <line:16:1, col:10> koo 'long ()'
```

```
|-FunctionDecl 0x6b511e0 prev 0x6b24910 <line:17:1, col:12> f 'int (int)'
| `-ParmVarDecl 0x6b51160 <col:7, col:11> x 'int'
|-VarDecl 0x6b512a0 <line:18:1, col:15> alal 'double'
| `-FloatingLiteral 0x6b512f8 <col:15> 'double' 1.000000e+00
|-FunctionDecl 0x6b51370 <line:19:1, col:35> foo 'void ()' static inline
| `-CompoundStmt 0x6b51430 <col:25, col:35>
|   `-ReturnStmt 0x6b51410 <col:27>
`-FunctionDecl 0x6b51470 <line:21:1, col:35> goo 'void ()' static inline
  `-CompoundStmt 0x6b51530 <col:25, col:35>
    `-ReturnStmt 0x6b51510 <col:27>
```

任务2 添加C语言对`#pragma elementWise`操作的支持

测试程序见当前目录下test1.c，测试方法详见老师提供的实验说明。

```c
//test1.c
#pragma elementWise
extern char joo();
int f(int x)
{
    int y = 2;
    int result = (x / 42);
    #pragma elementWise
    return result;
}
struct g{
    #pragma pack(1)
    #pragma elementWise
    int x;
    long aa;
};
long koo();
int f(int x);
double alal = 1.0;
static inline void foo(){ return; }
#pragma elementWise asd
static inline void goo(){ return; }
```

上面的代码中，joo和koo为函数声明，不应该标记。f有定义有声明，打印结果时应打印有标记的定义的结果。在结构体g和函数f中有elementWise制导，但按照要求被忽略并发出警告，所以foo不应该被标记上。goo不应被标记，因为其上的elementWise制导后有多余的asd符号，应该忽略并报警告。结果如下，符合预期。

```
./llvm-install/bin/clang -cc1 -load ./build/lib/TraverseFunctionDecls.so -plugin
traverse-fn-decls ./test1.c
./test1.c:7:13: warning: '#pragma elementWise' should be placed on top level -
ignored
    #pragma elementWise
            ^
./test1.c:12:13: warning: '#pragma elementWise' should be placed on top level -
ignored
    #pragma elementWise
            ^
./test1.c:20:21: warning: extra tokens at end of '#pragma elementWise' - ignored
#pragma elementWise asd
                    ^
```

```
f: 1
foo: 0
goo: 0
joo: 0
koo: 0
3 warnings generated.
```

更多的测试：

(1) 不含制导

```
#include "example1.h"
int main()
{ return 0; }
```

```
test result:
main: 0
```

测试结果正常

(2) 连续制导：一个制导只能匹配往后最近的函数，不会越过函数匹配。

```
#pragma pack (1)
void foo1()
{ return; }

#pragma elementWise
#pragma elementWise
void foo2()
{ return; }
void foo3()
{ return; }
```

```
test result:
foo1: 0
foo2: 1
foo3: 0
```

（3）制导与函数定义间加入函数声明及函数间的调用：

```
#pragma elementWise

int foo1();
int foo2();

int foo1(){
    return foo2();
}

int foo2(){
    int a = 1;
    int b = a - 3;
    return b;
}
```

```
int main(){
    return foo1();
}
```

```
test result:
foo1: 1
foo2: 0
main: 0
```

可以看到，中间的函数声明并不在制导范围内。函数中的函数调用对制导也无影响。

（4）制导与函数定义间加入宏定义：

```
#pragma elementWise
#define F(x)  x - 3

int foo2(){
    int a = 1;
    int b = a - 3;
    return F(b);
}

#pragma elementWise

#define m 5

int foo1(){
    int x = m;
    return foo2();
}

#pragma elementWise
#define iiii int

int main(){
    iiii x = 3;
    return foo1();
}
```

```
test result:
foo1: 1
foo2: 1
main: 1
```

可以看到，宏定义不会影响制导。

（5）改变函数类型：

```
#pragma elementwise

int foo2(){
    int a = 1;
    int b = a - 3;
    return b;
}
```

```
#pragma elementWise

char foo1(){
    return (foo2() > 0) ? '1' : '0';
}

#pragma elementWise
void foo3(){
    int x = 1;
}

int main(){
    return foo1();
}
```

```
test result:
foo1: 1
foo2: 0
foo3: 1
main: 0
```

举例改变函数类型为char和void时，正常制导。而foo1特意将elementWise大小写敲错，不能制导，验证区分大小写。

分成员总结

陈彦帆

主要负责实验报告撰写和修改部分代码。
顺着gdb摸了一遍clang前端从main函数到生成FunctionDecl的流程，终于厘清HandlePragma的调用流程。由于C++的指针特性和庞大的项目，不经过调试很难知道一些指针指向哪里。顺便恶补了一下git和gdb的使用。

骆嵩涛

主要负责则代码的撰写和修改部分报告。
按照老师给出的pragma pack的例子照猫画虎，首先将elementwise注册进preprocessor，之后给elementwise定义标记并在函数中加入elementwise这一属性。

毛啸腾

个人原因本次实验入手较晚，其他组员基本已经完成了本次实验。进行了实验结果的测试和相关报告部分增补，熟悉了项目代码的框架以及git和gdb等，通过理顺调用流程，在组员的帮助下相对容易地理解了本次实验，为进行之后的实验奠定了基础。