编译原理研讨课实验PR001实验报告

任务说明

进行clang和llvm环境配置，能够使用clang查看语法树
给llvm增加一个编译制导*#Pragma elementWise*.
暂时不用实现其具体功能，但是要求能够确认每个函数是否在制导范围之内。

成员组成

蔡昕，段江飞，资威

实验设计

设计思路

设计时候的思路就是理解一个编译制导是怎么工作的，每一步都需要什么代码去实现它。
具体来说就是跟着从词法分析读入#pragma开始，然后在词法分析中确认是elementWise。
最后进入对应的语义分析之中进行功能的实现。

实验实现

首先给出最后测试文件的AST语法树，证明我们llvm和clang的环境配置的没有问题。

```
[clang7@host2 ~]$ clang -Xclang -ast-dump -fsyntax-only test/test.c
In file included from test/test.c:1:
test/test.h:8:21: warning: extra tokens at end of '#pragma elementWise' - ignored
#pragma elementWise ok
                    ^
test/test.c:24:21: warning: extra tokens at end of '#pragma elementWise' - ignored
#pragma elementWise ok
                    ^
TranslationUnitDecl 0x5decb30 <<invalid sloc>>
|-TypedefDecl 0x5ded010 <<invalid sloc>> __int128_t '__int128'
|-TypedefDecl 0x5ded070 <<invalid sloc>> __uint128_t 'unsigned __int128'
|-TypedefDecl 0x5ded3c0 <<invalid sloc>> __builtin_va_list '__va_list_tag [1]'
|-FunctionDecl 0x5ded460 <test/test.h:2:1, col:8> g3 'int ()'
|-FunctionDecl 0x5ded520 <line:3:1, line:6:1> g2 'int ()'
| `-CompoundStmt 0x5ded6c0 <line:3:9, line:6:1>
|   |-DeclStmt 0x5ded648 <line:4:2, col:11>
|   | `-VarDecl 0x5ded5d0 <col:2, col:10> a 'int'
|   |   `-IntegerLiteral 0x5ded628 <col:10> 'int' 0
|   `-ReturnStmt 0x5ded6a0 <line:5:2, col:9>
|     `-ImplicitCastExpr 0x5ded688 <col:9> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x5ded660 <col:9> 'int' lvalue Var 0x5ded5d0 'a' 'int'
|-FunctionDecl 0x5ded710 prev 0x5ded460 <line:9:1, line:12:1> g3 'int ()'
| `-CompoundStmt 0x5e1a170 <line:9:9, line:12:1>
|   |-DeclStmt 0x5ded838 <line:10:2, col:11>
|   | `-VarDecl 0x5ded7c0 <col:2, col:10> a 'int'
|   |   `-IntegerLiteral 0x5ded818 <col:10> 'int' 0
|   `-ReturnStmt 0x5e1a150 <line:11:2, col:9>
|     `-ImplicitCastExpr 0x5e1a138 <col:9> 'int' <LValueToRValue>
|       `-DeclRefExpr 0x5e1a110 <col:9> 'int' lvalue Var 0x5ded7c0 'a' 'int'
|-FunctionDecl 0x5e1a1c0 <line:16:1, line:19:1> g4 'int ()'
| `-CompoundStmt 0x5e1a360 <line:16:9, line:19:1>
|   |-DeclStmt 0x5e1a2e8 <line:17:2, col:11>
|   | `-VarDecl 0x5e1a270 <col:2, col:10> a 'int'
```

这里就以看到一个**#pragma elementWise**怎么处理来解释我们是怎么具体实现的。

```
77 void PragmaNamespace::HandlePragma(Preprocessor &PP,
78                                     PragmaIntroducerKind Introducer,
79                                     Token &Tok) {
80   // Read the 'namespace' that the directive is in, e.g. STDC.  Do not macro
81   // expand it, the user can have a STDC #define, that should not affect this.
82   PP.LexUnexpandedToken(Tok);
83
84   // Get the handler for this token.  If there is no handler, ignore the pragma.
85   PragmaHandler *Handler
86     = FindHandler(Tok.getIdentifierInfo() ? Tok.getIdentifierInfo()->getName()
87                                           : StringRef(),
88                   /*IgnoreNull=*/false);
89   if (Handler == 0) {
90     PP.Diag(Tok, diag::warn_pragma_ignored);
91     return;
92   }
93
94   // Otherwise, pass it down.
95   Handler->HandlePragma(PP, Introducer, Tok);
96 }
```

如果词法识别到了#pragma之后就会调用这个函数，在82行获得下一个token，然后在84行寻找对应的Handler，所以我们首先要实现的就是把Handler设置好。我们首先定义了PragmaElementWiseHandler。

```
55 class PragmaElementWiseHandler : public PragmaHandler {
56 public:
57   explicit PragmaElementWiseHandler() : PragmaHandler("elementWise") {}
58
59   virtual void HandlePragma(Preprocessor &PP, PragmaIntroducerKind INtroducer,
60                             Token &FirstToken);
61 };
```

接着定义OwningPtr的一个实例，用于Parser里注册elementWise制导的处理函数。

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

最后在Parser的构造和析构函数中注册一下，使得其会被预处理，在最后也会被清除。

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

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

这样下图中PragmaNamespace::HandlePragma中调用的FindHandler就可以找到对应得Handler并且调用Handler->HandlePragma。所以我们接下来需要实现这个函数。

下面就是Handler->HandlePragma.

```
370 // #pragma elementWise
371 void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
372                                              PragmaIntroducerKind Introducer,
373                                              Token &elementWiseTok) {
374   //TODO:student coding here
375   Sema::PragmaElementWiseKind Kind = Sema::EWise_OFF;
376
377   Token Tok;
378   PP.Lex(Tok);
379   if (Tok.isNot(tok::eod)) {
380     PP.Diag(Tok.getLocation(), diag::warn_pragma_extra_tokens_at_eol) << "elementWise";
381     return ;
382   }
383
384   Kind = Sema::EWise_ON;
385   Token *Toks = (Token *) PP.getPreprocessorAllocator().Allocate(
386                           sizeof(Token) * 1, llvm::alignOf<Token>());
387   new (Toks) Token();
388   Toks[0].startToken();
389   Toks[0].setKind(tok::annot_pragma_elementWise);
390   Toks[0].setLocation(elementWiseTok.getLocation());
391   Toks[0].setAnnotationValue(reinterpret_cast<void*>(
392                              static_cast<uintptr_t>(Kind)));
393   PP.EnterTokenStream(Toks, 1, /*DisableMacroExpansion=*/true,
394                       /*OwnTokens=*/false);
395
396 }
```

这个地方的功能是看一看是不是真的elementWise编译制导，其实就是判断了一下elementWise后面有没有接着别的字符，如果有那就要报错了(由379-382行实现)，如果没有就需要把信息放在一个Token里面传递给Sema.这里就需要说明一下Token的初始化和定义。

```
613 // Annotation for #pragma elementWise...
614 // The lexer produces these so that they only take effect when the parser
615 // handles them.
616 ANNOTATION(pragma_elementWise)
617
```

首先定义一个Token。

```
enum PragmaElementWiseKind {
  EWise_OFF, // #pragma elementWise
  EWise_ON
};
```

接着就是给出可能的两种语义信息，也就是在此编译制导范围内或者不在。接下来我们就可以再回到刚才那个函数。

```
370 // #pragma elementWise
371 void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
372                                     PragmaIntroducerKind Introducer,
373                                     Token &elementWiseTok) {
374   //TODO:student coding here
375   Sema::PragmaElementWiseKind Kind = Sema::EWise_OFF;
376
377   Token Tok;
378   PP.Lex(Tok);
379   if (Tok.isNot(tok::eod)) {
380     PP.Diag(Tok.getLocation(), diag::warn_pragma_extra_tokens_at_eol) << "elementWise";
381     return ;
382   }
383
384   Kind = Sema::EWise_ON;
385   Token *Toks = (Token *) PP.getPreprocessorAllocator().Allocate(
386                           sizeof(Token) * 1, llvm::alignOf<Token>());
387   new (Toks) Token();
388   Toks[0].startToken();
389   Toks[0].setKind(tok::annot_pragma_elementWise);
390   Toks[0].setLocation(elementWiseTok.getLocation());
391   Toks[0].setAnnotationValue(reinterpret_cast<void*>(
392                              static_cast<uintptr_t>(Kind)));
393   PP.EnterTokenStream(Toks, 1, /*DisableMacroExpansion=*/true,
394                       /*OwnTokens=*/false);
395
396 }
```

可以看到在375行首先默认是没有编译制导的，在确认制导正确之后再384行改为有编译制导，然后定义一个Token准备把信息传给Sema。后面的代码都是初始话对应的Token。

接下在就到了语义部分。

首先要给一个全局变量，让Sema存储是否有elementWise制导。

```
/// IsElementWise - status for ElementWise
bool IsElementWise;
```

然后这个变量会在函数定义的构造函数中初始化：

```
FunctionDecl(Kind DK, DeclContext *DC, SourceLocation StartLoc,
             const DeclarationNameInfo &NameInfo,
             QualType T, TypeSourceInfo *TInfo,
             StorageClass S, bool isInlineSpecified,
             bool isConstexprSpecified)
  : DeclaratorDecl(DK, DC, NameInfo.getLoc(), NameInfo.getName(), T, TInfo,
                   StartLoc),
    DeclContext(DK),
    ParamInfo(0), Body(),
    SClass(S), IsElementWise(0),
```

同时我们需要**set**和**get**函数来访问它。

```
1617   void setIsElementWise(bool IsElementWise);
1618   bool getIsElementWise() const {return IsElementWise;};
1619
```

```
2036 void FunctionDecl::setIsElementWise(bool ElementWiseRule) {
2037   assert(doesThisDeclarationHaveABody());
2038   IsElementWise = ElementWiseRule;
2039 }
```

那我们现在要去用Token了。在处理函数定义的时候它会看到我们的Token，然后调用HandlePragmaElementWise。

```
609 Parser::DeclGroupPtrTy
610 Parser::ParseExternalDeclaration(ParsedAttributesWithRange &attrs,
611                                   ParsingDeclSpec *DS) {
612   DestroyTemplateIdAnnotationsRAIIObj CleanupRAII(TemplateIds);
613   ParenBraceBracketBalancer BalancerRAIIObj(*this);
614
615   if (PP.isCodeCompletionReached()) {
616     cutOffParsing();
617     return DeclGroupPtrTy();
618   }
619
620   Decl *SingleDecl = 0;
621   switch (Tok.getKind()) {
622   case tok::annot_pragma_vis:
623     HandlePragmaVisibility();
624     return DeclGroupPtrTy();
625   case tok::annot_pragma_pack:
626     HandlePragmaPack();
627     return DeclGroupPtrTy();
628   case tok::annot_pragma_elementWise:
629     HandlePragmaElementWise();
630     return DeclGroupPtrTy();
```

下面这个函数的功能就是首先检查是不是正确的Token，然后消耗掉这个Token，把信息递给ActOnPragmaElementWise。

```
65 void Parser::HandlePragmaElementWise() {
66   assert(Tok.is(tok::annot_pragma_elementWise));
67   Sema::PragmaElementWiseKind Kind =
68   static_cast<Sema::PragmaElementWiseKind>(
69   reinterpret_cast<uintptr_t>(Tok.getAnnotationValue()));
70   Actions.ActOnPragmaElementWise(Kind);
71   ConsumeToken();
72 }
```

它的功能就是给IsElementWise赋值。

```
262 void Sema::ActOnPragmaElementWise(PragmaElementWiseKind Kind) {
263   IsElementWise = (Kind == EWise_ON);
264 }
```

其它

然后说一下我们是怎么测试的。下面是我们的测试文件

下面这个是**test.c**

```c
#include "test.h"

int f(int x){
  int result = (x / 42);
  return result;
}


#pragma elementWise
int f1()
{
  int a = 0;
  return a;
}

#pragma elementWise
int f3();
int f2()
{
  int a = 0;
  return a;
}

#pragma elementWise ok
int f3()
{
  int a = 0;
  return a;
}

#pragma elementWise
#pragma elementWise
int f4()
{
  int a = 0;
  return a;
}
int f5()
{
  int a = 0;
  return a;
}

```

```c
#pragma elementWise
int b;
int c;
int f6();
```

下面这个是**test.h**

```
 1 #pragma elementWise
 2 int g3();
 3 int g2(){
 4   int a = 0;
 5   return a;
 6 }
 7
 8 #pragma elementWise ok
 9 int g3(){
10   int a = 0;
11   return a;
12 }
13
14 #pragma ElementWise
15 #pragma ElementWise
16 int g4(){
17   int a = 0;
18   return a;
19 }
20 int g5(){
21   int a = 0;
22   return a;
23 }
24
```

可以看到test.c是我们的主体，其中**include**了test.h。

在test.c里面我们可以看到f1是正常的应该在制导范围内的，f3是只是声明，所以f2应该在制导范围内而f3不在。然后第24行elementWise后面加了别的东西，这个地方应该报error。然后重复的编译制导依然之后后面那一个有效，所以只有f4会在编译制导范围内，同时f6是声明不是定义所以也不在编译制导范围内。在这个test文件中我们测试了定义和申明的区分，编译制导和函数定义中有别的东西以及elementWise后面非换行符的情况。

然后就是**include**的文件，这里大同小异测试了类似的内容。多了一个测试内容，使用了ElementWise，这里应该直接识别不到才对，最后测试结果是这样的：

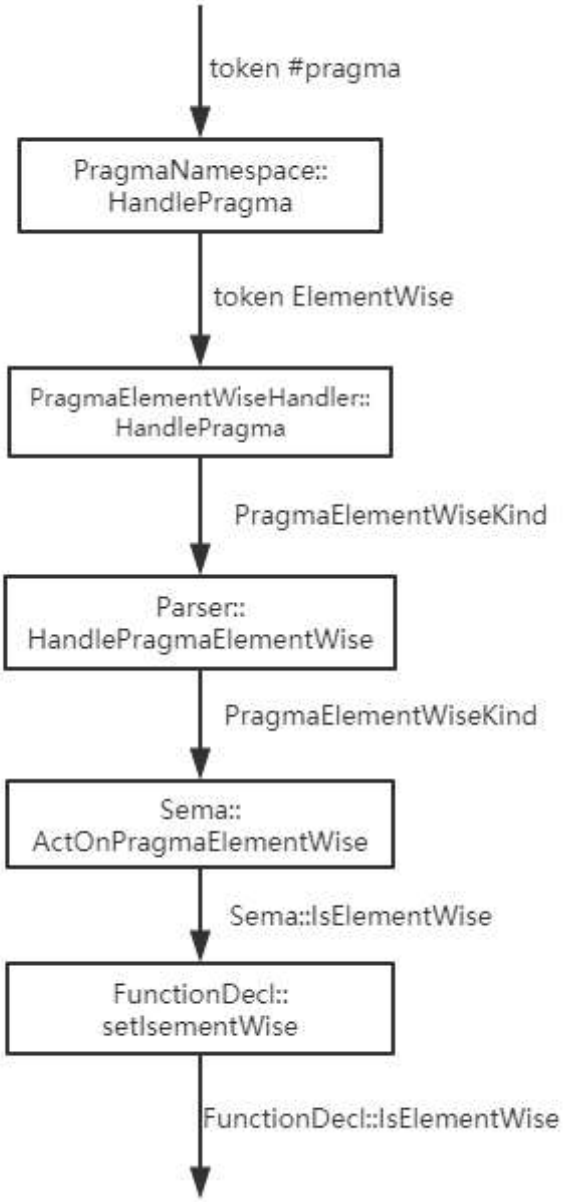
```
[clang7@host2 ~]$ sh run.sh test/test.c
In file included from test/test.c:1:
test/test.h:8:21: error: extra tokens at end of '#pragma elementWise' - ignored
#pragma elementWise ok
                    ^
test/test.c:24:21: error: extra tokens at end of '#pragma elementWise' - ignored
#pragma elementWise ok
                    ^
f: 0
f1: 1
f2: 1
f3: 0
f4: 1
f5: 0
f6: 0
g2: 1
g3: 0
g4: 0
g5: 0
```

总结

实验结果总结

修改的文件汇总如下：
./tools/clang/lib/Parse/ParsePragma.h
./tools/clang/lib/Parse/ParsePragma.cpp
./tools/clang/lib/Parse/Parser.cpp
./tools/clang/include/clang/Parse/Parser.h
./tools/clang/include/clang/Basic/TokenKinds.def
./tools/clang/include/clang/Sema/Sema.h
./tools/clang/lib/Sema/SemaDecl.cpp
./tools/clang/lib/Sema/SemaAttr.cpp
./tools/clang/lib/Sema/Sema.cpp
./tools/clang/include/clang/AST/Decl.h
./tools/clang/lib/AST/Decl.cpp
./tools/clang/lib/Lex/Pragma.cpp
./tools/clang/examples/TraverseFunctionDecls/TraverseFunctionDecls.cpp

在这次实验中，我们把制导信息从最初的字符串一步一步解析成了函数声明的属性，以下是流程示意图：



在此次实验中我们遇到了一个bug，那就是一开始没有在FunctionDecl中初始化IsElementWise，它应该初始化为0，代表在最开始默认并没有在elementWise的编译制导之中。

分成员总结

蔡　昕：完成了实验中进行语义分析处理部分(即从Token流中的tok::annot_pragma_elementWise的Token到Sema中的IsElementWise的状态，再从IsElementWise到函数的定义的信息传递的过程)的代码，以及对输出结果的函数(TraverseFunctionDecls.cpp)的修改，并参与了部分调试和实验报告的工作。

段江飞：环境配置，完成了从词法分析到生成Token流的代码，进行调试，输出脚本。

资　威：实验开头的大体框架，进行结果验证，实验报告。