编译原理研讨课实验PR001实验报告

任务说明

修改llvm源码使得它支持添加了element wise的#pragma编译制导，该制导实现编译时打印函数名称，并显示该函数是否在制导范围内.若在，则打印"函数名: 1",否则打印"函数名:0".

成员组成

陈灿宇
宋鹏皓
金越

实验设计

设计思路

总体思路：识别并正确得到编译制导结果需要llvm中的Preprocessor/Lexer/Parser/Sema协同工作。我们需要定义相关的pragma处理函数以及与并能让编译器识别elementWise标识符。

1. **Preprocessor**的工作

   需要添加和注册相应的处理类实例，以便Lex识别到对应的`elementWise`标识时，能够转到到对应处理函数。

2. **Lexer**的工作：在**Lexer**中识别**Pragma**

   当词法处理器Lexer认出`#Pragma`时会调用`HandlePragma（Preprocessor &PP, PragmaIntroducerKind Introducer, Token &Tok）`函数处理这个`Pragma`，Lexer会根据`Token`来寻找对应的`Handler`，之后，再调用`Handler`的`HandlePragma`函数处理相应的`Pragma`。

   因此，我们定义处理`element wise`的`Handler`并实现其对应的`HandlePragma`函数。

3. **Parser**的工作：在**Parser**中定义**Handler**

   Parser的工作：具体完成Handler的处理函数实现，这个处理函数主要做的工作就是判断对应Token的合法性，并把它变成一个程序能辨别的Token，然后把它加入到Token流中传递给后续的Parser和Sema处理。后续的Parser需要实现一个HandleParagmaElementWise函数用来接受前面传递过来的token，然后消耗此token处理下一个token，之后调用Sema完成对"elementWise"对应token的响应。

   对于Handler的定义是在Parser中进行的。根据我们的需求，我们要在`ParsePragma.h`中定义一个`PragmaElementWiseHandler`，并在`Parser.h`中定义一个Hander实例，同时，在`Parser.cpp`中的析构函数`~Parser()`中加上对应的处理。

   接着，我们需要在`ParsePragma.cpp 中`实现`PragmaElementWiseHandler`的`HandlePragma`的核心逻辑，即如何对`#pragma element wise`做具体的处理。首先，判断是否为合法的`element wise` Pragma，按照我们的要求，合法的Pragma只能是`#pragma element wise` 。如果确认Pragma合法， 将这个信息记录下来，记录为一个`tok::annot_pragma_element_wise`Token，传递给后续的Parser/Sema进行处理。

后续的处理过程，由Parser承担的是，在`Parser.cpp`中的`ParseExternalDeclaration()`函数中根据Token类型调用相应的Handle函数，对于`element wise`，这个Handle函数是定义在 在`ParsePragma.cpp`中的`HandlePragmaElementWise()`，这个Handle函数将Token通过`ActOnPragmaElementWise()`函数传递给Sema过程。`ActOnPragmaElementWise()`是定义在Sema中的函数，我们会在下一部分讲到。

4. **Sema的工作：在Sema中处理Token**

   首先，对于一个Token往往要记录其对应的语义信息，对应于一个Pragma的规则几，这一语义信息`PragmaKind`在`Sema.h`中定义，但由于本次实验的`element wise`不存在额外的语义信息，所以我们跳过这一步。

   然后，对于语义分析的过程，我们需要记录下来当前`IsElementWise`的状态，即，此时如果出现一个函数的定义，那么`IsElementWise`的状态是什么？

   然后，我们还需要修改一下函数定义的属性，标记其对应的`IsElementWise`状态，并增加相应的`Set`和`Get`函数。

   现在，我们要在Sema中添加对于Token的处理函数。在之前的Parser中调用了`ActOnPragmaElementWise()`函数，现在我们需要到`SemaAttr.cpp`中去实现这个函数，需要做的，就是把`IsElementWise`置为1。

   除此之外，我们还需要到`SemaDecl.cpp`中修改`ActOnFinishFunctionBody()`函数，根据当前的`IsElementWise`状态为函数体设置`IsElementWise`状态，并把`IsElementWise`置0，因为一个Pragma最多只能匹配一个函数定义。

5. 输出函数定义

   这一部分代码已经由老师提供。

实验实现

> 1. `llvm-3.3/tools/clang/lib/Parse/ParsePragma.h`中添加PragmaElementWiseHandler类，定义其构造方法（给基类PragmaHandler传递其名字参数"elementWise"）

```
/// PragmaElementWiseHandler - "\#pragma elementWise"
class PragmaElementWiseHandler : public PragmaHandler {
    public:
    PragmaElementWiseHandler() : PragmaHandler("elementWise") {}
    virtual void HandlePragma(Preprocessor &PP, PragmaIntroducerKind Introducer,
                      Token &FirstToken);
};
```

> 2. 将定义的handler添加给预处理器Preprocessor,
>
> 1). `llvm-3.3/tools/clang/include/clang/Parse/Parser.h 153`中定义elementWiseHandler的实例：

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

> 2). `llvm-3.3/tools/clang/lib/Parse/Parser.cpp 66` 在Parser的构造函数里添加Handler：

```
// Add #pragma handlers. These are removed and destroyed in the
// destructor.
ElementWiseHandler.reset(new PragmaElementWiseHandler());
PP.AddPragmaHandler(ElementWiseHandler.get());
```

3). `llvm-3.3/tools/clang/lib/Parse/Parser.cpp 66` 在Parser的析构函数里添加Handler移除：

```
// Remove the pragma handlers we installed.
PP.RemovePragmaHandler(ElementWiseHandler.get());
ElementWiseHandler.reset();
```

3. `tools/clang/include/clang/Basic/TokenKinds.def 665` 中，定义一个elementWise对应的Token:

```
ANNOTATION(pragma_element_wise)
```

4. 在`tools/clang/lib/Parse/ParsePragma.cpp 870`中，添加设计中所述的Parser的前序工作：即判断合法性（这里就看下一个token是否是EOF，若不是则为非法的编译制导elmentwise）然后仿照其他pragma的做法，把token塞到token流里传给后续Parser.此时我们需要处理的token的kinde已经变成tok::annot_pragma_element_wise.这就是后续Parser和Sema处理这个Token所需要识别的东西。

```
/// \brief Handle '#pragma elementWise' before a Function.
///
void PragmaElementWiseHandler::HandlePragma(Preprocessor &PP,
                                            PragmaIntroducerKind Introducer,
                                            Token &ElementWiseTok) {

    Token Tok;
    PP.Lex(Tok);
    /// now to check pragma elementwise
    if (Tok.isNot(tok::eod)) {// end of line
        PP.Diag(Tok.getLocation(), diag::warn_pragma_extra_tokens_at_eol) << "elementwise";
        return;
    }

    Token *Toks =
        (Token*) PP.getPreprocessorAllocator().Allocate(
        sizeof(Token) * 1, llvm::alignOf<Token>());
    new (Toks) Token();
    Toks[0].startToken();
    Toks[0].setKind(tok::annot_pragma_element_wise);
    Toks[0].setLocation(ElementWiseTok.getLocation());

    PP.EnterTokenStream(Toks, 1, true, false );
}
```

5. `tools/clang/include/clang/Sema/Sema.h 268`我们需要在Sema里定义一个变量记录elementWise的状态。依此来设置函数声明里函数的elementWise状态。

```
bool IsElementWise;
```

6. `tools/clang/include/clang/AST/Decl.h`中，我们需要在函数声明（定义）里也设置一个变量来记录elementWise状态，并设置相应的get和set函数来获取和设置这个状态，这个变量是一个一位bool类型变量，为1表示该函数在`#pragma elementWise`制导范围内，为0表示不在。而set和get函数实现比较简单，这里直接实现在类声明里。

```
bool IsElementWise : 1;
//...此处省略很多无关代码
void setIsElementWise(bool isElementWise){IsElementWise = isElementWise;}
bool getIsElementWise() const { return IsElementWise; }
```

7. 在4中我们定义了前序Parser需要做的事，此处我们定义后序Parser需要做的，后序Parser从前序传递来Token流中碰到*tok:annot_pragma_element_wise*，我们需要对其进行处理，因此需要在`tools/clang/include/clang/Parse/Parser.h 422`定义处理函数：

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

该函数在`tools/clang/include/clang/Parse/Parser.cpp`中实现：

```
void Parser::HandlePragmaElementWise() {
    assert(Tok.is(tok::annot_pragma_element_wise));
    ConsumeToken(); // The annotation token.
    Actions.ActOnPragmaElementWise();
}
```

8. 7中ActOnPragmaElementWise函数在`tools/clang/include/clang/Sema/Sema.h 2724`中定义：

```
void ActOnPragmaElementWise();
```

该函数实现在`tools/clang/lib/Sema/SemaAttr.cpp 366`中，它更改Sema内部的IsElementWise状态为1：

```
void Sema::ActOnPragmaElementWise(){
    IsElementWise = 1;
}
```

9. 我们还需要为`HandlePragmaElementWise`函数在Parser添加调用：`tools/clang/lib/Parse/Parser.cpp 653`中，在`Parser::ParseExternalDeclaration`函数那一大堆对token的kinde进行判断switch-case语句里添加其调用：

```
switch (Tok.getKind()) {
//省略无关代码
case tok::annot_pragma_element_wise:
    HandlePragmaElementWise();
    return DeclGroupPtrTy();
}
```

10. 上述过程完成了Sema记录elementWise状态的过程，接下来我们需要把这个状态传递给函数声明。在`tools/clang/lib/Sema/SemaDecl.cpp 8881`中添加我们前面定义函数声明FD的element-set函数。它将Sema记录的elemtnWise状态传递给FD。

```
Decl *Sema::ActOnFinishFunctionBody(Decl *dcl, Stmt *Body,bool IsInstantiation) {
    FunctionDecl *FD = 0;
        //省略无关代码
    if (FD) {
        FD->setBody(Body);
        FD->setIsElementWise(IsElementWise);
        IsElementWise = 0;
    }
}
```

> 11. 准备测试.c文件，放在`home/test/`下。

其它

总结

实验结果总结

> 测试结果前的准备：从clang0下把elementWise用的TraverseFunctionDecls打印函数拷贝到相应目录里，重复编译安装`llvm/clang`，编译安装Plugin。
> 执行命令：`~/llvm/bin/clang -cc1 -load ~/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/test/P1_test1.c (or P1_test2.c, P1_test3.c, P1_test4.c, P1_test_error.c) -Werror` 或者 `sh /home/clang9/PR001/scripts/compile_and_check.sh P1_test1.c (or P1_test2.c, P1_test3.c, P1_test4.c, P1_test_error.c)`
> 测试结果如下：

- 实验一：

  测试程序：P1_test1.c

  ```
  #pragma elementWise
  int main(){
      return 0;
  }
  int foo(){
      return 1;
  }
  ```

  实验结果：

  ```
  foo: 0
  main: 1
  ```

  实验证明，我们的编译器能够识别正常情况下的pragma，并且每个pragma只匹配一个函数定义。

- 实验二

  测试程序：P1_test2.c

```
#pragma elementWise
int main(){
    return 0;
}
#pragma elementWise
void stupid();

int foo(){
    return 1;
}
```

实验结果：

```
foo: 1
main: 1
stupid: 0
```

测试程序：P1_test3.c

```
#pragma elementWise
int main(){
    return 0;
}
#pragma elementWise
void stupid();

int foo(){
    return 1;
}

void stupid(){
    return;
}
```

实验结果：

```
foo: 1
main: 1
stupid: 0
```

这次实验的两个程序说明，在Pragma和函数定义之间插入函数声明，并不会给声明的函数加上Pragma，同时也不会影响到给后面的函数加上Pragma。

- 实验三

  测试程序：P1_test4.c

```
#pragma elementWise
int main(){
    return 0;
}

int foo(){
    return 1;
}
#pragma elementWise
int i = 0;
int k = 1;
void stupid(){
    return;
}
```

实验结果：

```
foo: 0
main: 1
stupid: 1
```

本次实验说明，Pragma跟函数定义之间插入的赋值语句并不影响给函数加上Pragma。

- 实验四

  测试程序：P1_test_error.c

```
#pragma elementWise 1
int foo()
{
    return 1;
}
```

实验结果：

```
/home/clang9/test/P1_test_error.c:1:21: error: extra tokens at end of '#pragma elementwise'
- ignored
#pragma elementWise 1
                    ^
foo: 0
1 error generated.
```

本次实验说明，编译时期的输出(elementWise的检查结果)放到标准错误上。

分成员总结

> 组员： 陈灿宇

本次实验真正让我了解了编译器是如何工作的，在理论课上学习的知识比较抽象，真正动手做才能够比较深刻地理解。实验开始的时候进行得比较困难。老师发布的Pragma说明，以AsCheck为例，比较详细的解释了Pragma在程序中处理的整个流程，因此才比较容易上手。 但对于elementwise而言，不需要处理各种kind，所以相对而言，比ascheck简单一些。

> 组员：金越

这次的实验虽然看上去很庞大的样子，但是实际真正做下来发现要做的东西并不太多，是因为之前看代码时发现llvm已有的几个pragma编译制导选项（例如pack）在关键token后面都会有更多的选项。但我们需要实现的elementWise本质上其实只是更改一个状态变量。实验前由于对C++不熟练还去专门补习了一下C++，后来发现实验对C++的要求其实很高，假如没有老师给的实现说明文档，自己单独实现还是比较困难的。我觉得实验的难点就在于想清楚要实现pragrama编译制导这样一个功能，我们需要更改什么？ 一个朴素的想法是类比，由于很多pragma编译制导都是围绕函数定义来做的，因此实际上可以参考某些函数定义时候的关键字（例如inline、static等。看完llvm源码后发现这些关键字的实现其实都是更改函数定义类里的状态。那么问题又变成如何识别这些关键字对应的token,当然这个可以类比其他pragma处理得到。所以类比的思想在这次实验设计和实现（尤其是功能定位）起到很大的作用。

> 组员：宋鹏皓

本次实验提供了一个阅读大量源码的机会，llvm本身的代码比较繁多，如果直接下手的话难免有些不知所措，但同时这也是一个锻炼自己阅读大型代码的能力。所以在研究要实现的各个关键函数的过程中，对于不认识的函数或者类，必要时进行追根究底的查找定义需要花费很多功夫，不过这也帮助了我对C++这个语言更加了解，毕竟之前的学习中对C++也并不是特别熟悉。此外，通过本次实验，借助老师提供的《Pragma实现说明》，理论课上的许多抽象的知识有了具体的实现，这使得我对编译有了更深的理解，也更清晰地明白了编译器的工作流程。总的来说，本次实验逻辑并不复杂，如果理清了编译过程，那么各个部分也就水到渠成了。

成员贡献

> 组员：陈灿宇 ：独立实现最终版的全部代码，帮助完善报告
> 组员：金越 ：撰写报告，参与讨论，独立实现代码
> 组员：宋鹏皓 ：思考并参与讨论，独立实现代码