编译原理研讨课实验PR001说明

熟悉Clang的安装和使用

编译安装LLVM和Clang：

第一步：登录到本组服务器的帐号，参见实验环境说明。

每个组有两个账号密码组合：

1. 服务器登录：`llvm1:llvm1!`，其中`1`为组号；
2. Gitlab账号：`llvm1:llvm1!llvm1!`，其中`1`为组号；

第二步：将源代码从Gitlab服务器clone到本地：

配置自己的git偏好设置:

```
git config --global user.name "llvm1 1"
git config --global user.email "llvm1xxx@xxxx.com"
git config --global core.editor vim
# vim or emacs, up to you
```

拷贝代码：

```
git clone http://PowerEdge-M640-Blade-8/llvm1/llvm.git
# '1' is your group number
```

输入Gitlab账号和密码，形如：`llvm1:llvm1!llvm1!`

第三步：编译和安装LLVM和Clang

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/bin/gcc && export CXX=/usr/bin/g++`
2. 生成Makefile：`cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=~/llvm-install ../llvm`，其中*CMAKE_INSTALL_PREFIX*代表了编译完成以后的安装目录，*../llvm*代表了源代码目录
3. 使用*make*进行编译：`make -j15`，其中*j15*代表使用15个线程并行编译。该过程约需要数分钟。如果仅仅是更改了*Clang*的源代码 （也就是大家做作业的过程当中），则无需执行第二步，。
4. 安装到`CMAKE_INSTALL_PREFIX`当中：`make install`
5. 检查`~/llvm-install`下是否已经有对应的文件安装进去：`~/llvm-install/bin/clang --version`，将会有`clang version 3.3 (tags/RELEASE_33/final)`对应字样输出

生成和查看C程序对应的AST

第一步，准备一个简单的C程序`test.c`：

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

第二步，使用`clang`将AST给dump出来：`~/llvm-install/bin/clang -Xclang -ast-dump -fsyntax-only test.c`

使用GDB调试Clang

由于我们是使用GNU的*gcc*和*g++*编译生成的*Clang*，这意味着需要使用gdb来对*Clang*进行跟踪调试。当然，如果你使用的是LLVM编译生成*Clang*，对应的调试工具将是*lldb*。

调试*Clang*的典型流程：

1. 打开gdb：`gdb`
2. 打开要调试的*clang*可执行文件，通过*file*命令：`file ~/llvm-install/bin/clang`，将会有`Reading symbols from /home/clang1/llvm-install/bin/clang...done`字样的输出。
3. 设定调试子进程，因为*Clang*会派生一个新进程来执行编译流程，命令：`set follow-fork child`
4. 在处理Pragma的入口函数处打断点：`b clang::PragmaNamespace::HandlePragma`
5. 执行编译：`r example.c`，其中*example.c*是传递给*Clang*的参数
6. 进行正常调试

常见命令介绍：

`r args`，运行已经加载的应用，*args*是传递给应用的参数。

`l`，列出代码

`b`，在指定位置打断点

`c`，继续执行程序

输出插件使用说明

遍历函数定义

第一步，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。

第四步，执行。`~/llvm-install/bin/clang -cc1 -load ~/build/lib/TraverseFunctionDecls.so -plugin traverse-fn-decls ~/example.c`。同学们可以使用这一行命令来判断自己PR001完成的如何。

这里涉及的核心代码在：

`file: tools/clang/examples/TraverseFunctionDecls/TraverseFunctionDecls.cpp`

```
class TraverseFunctionDeclsVisitor
    : public RecursiveASTVisitor<TraverseFunctionDeclsVisitor> {
public:
    explicit TraverseFunctionDeclsVisitor(ASTContext *Context)
        : Context(Context) {}

    bool TraverseDecl(Decl *DeclNode) {
        if (DeclNode == NULL) {
```

```
            return true;
        }
        if (const FunctionDecl *FD = dyn_cast<FunctionDecl>(DeclNode)) {
            std::string name = FD -> getNameAsString();
            unsigned rule = FD -> getAsCheckRule();
            // 获取AsCheckRule，同学们可以实现类似的函数获得Element Wise的状态
            // 支持Element Wise操作的，输出1，不支持则输出0
            if(rule != 0) {
                funcNamesToAsCheckRule[FD->getNameAsString()] = FD ->
getAsCheckRule();
            } else {
                std::map<std::string, unsigned>::iterator it =
funcNamesToAsCheckRule.find(name);
               if(it == funcNamesToAsCheckRule.end())
                  funcNamesToAsCheckRule[FD->getNameAsString()] = FD ->
getAsCheckRule();
            }
        }
        return
RecursiveASTVisitor<TraverseFunctionDeclsVisitor>::TraverseDecl(DeclNode);
    }
    // 在TraverseFunctionDeclsConsumer中被调用，输出结果
    void OutputAsCheckRules() {
        for(std::map<std::string, unsigned>::iterator it =
funcNamesToAsCheckRule.begin(); it != funcNamesToAsCheckRule.end(); ++it) {
            llvm::outs() << it -> first << ": " << it -> second << "\n";
        }
        // 请不要改动输出格式，这将成为判分依据
        // 支持Element Wise操作的，输出1，不支持则输出0
    }
private:
    ASTContext *Context;
    // 内部代码，记录下每个函数对应的AsCheckRule的状态
    std::map<std::string, unsigned> funcNamesToAsCheckRule;
};
```