LLVM 学习

题目：

一、编写一个程序，将一个.ll 文件的所有加法更改为减法，该文件编译为可执行文件，能正常运行。

二、编写一个程序，在一个.ll 文件中的所有加法后面增加一个 if 语句块，当加法数值大于 100 时，数值减小 100，当加法数值小于 100 时，数值不变。该文件编译为可执行文件，能正常运行。

三、编写一个程序，在一个.ll 文件中增加一个函数，传递一个 int 类型参数，该函数判断传递的参数是否大于 100，大于 100 则自减 100，否则不变。函数返回处理后的数值。并且，在该.ll 文件的所有加法后面增加一条该函数的调用语句，对加法数值进行处理。并该文件编译为可执行文件，能正常运行。

四、编写一个程序，使其在 c-fft 的相关文件中完成插桩。完成插桩后，将 c-fft 编译为可执行文件，运行 c-fft，插桩代码将 c-fft 的函数调用树打印到一个文件中。

五、编写一个程序，使其在 C-Thread-Pool-master 的相关文件中完成插桩。完成插桩后，将 C-Thread-Pool-master 编译为可执行文件，运行 C-Thread-Pool-master，插桩代码将 C-Thread-Pool-master 的函数调用树打印到一个文件中。【注：多线程】

六、编写一个程序，实现对程序的 flow-sensitive、inter-procedure 分析，以适当的方式打印出数据流。

七、编写一个程序，实现对程序的静态函数调用图分析，并以适当的方式将该图输出到文件中。

八、编写一个程序，实现简易的 Anderson 指针分析算法，并在样例上测试。

九、编写一个程序，实现简易的 Steensgaard 指向分析算法，并在样例上测试。

十、编写一个简易版本的 fuzzing 工具，能够 fuzz 样例代码。Fuzzer 应当有：① 简单的 llvm 插桩工具，能够在目标程序的代码中插入代码使得能够在运行时提取分支覆盖率的信息；②简单的种子变异策略能够根据已有的种子随机生成不一样的输入；③能根据分支覆盖率的反馈自动调整种子生成、选取与变异，使得 fuzzer 能更快覆盖到更多的分支。

1. 题目一思路：

方法一：

1. 首先，需要检索出每一条 add 指令，因此需要对一个 module（.ll 文件）内的所有 instruction 进行遍历,具体可先对 module 内的每个 function 进行遍历,再对 function 内的每个 block 进行遍历，最后对 block 内的每个 instruction 进行遍历。逐个判断（getOpcode()）其操作类型是否是加法（Instruction::Add)。
2. 若当前指令为 add 指令，在其前面加入操作数相同的 sub 指令（BinaryOperator::Create)，并删除原有的 add 指令（eraseFromParent()）。
3. 完成 add 指令到对应 sub 指令的变更后,需要注意还需要修改跟随在 add 指令后的 store、ret 等指令。修改 store 指令时需使其保持存储地址不变，将其存储数据由 add 指令数据更改为 sub 指令数据，处理方法与 2.相似。
4. 完成对 handler.cpp 的编辑后,还需要对 main.cpp 进行修改,将 module 写入到文件中，具体实现可参照 llvm-dis 中的相关内容。

方法二：

1. 首先，需要检索出每一条 add 指令，因此需要对一个 module（.ll 文件）内的所有 instruction 进行遍历,具体可先对 module 内的每个 function 进行遍历,再对 function 内的每个 block 进行遍历，最后对 block 内的每个 instruction 进行遍历。逐个判断（getOpcode()）其操作类型是否是加法（Instruction::Add)。
2. 若当前指令为 add 指令，在其前面加入操作数（getOperand()获取）相同的 sub 指令，然后调用原指令的 replaceAllUsesWith()方法，把所有对 add 指令的引用改为对 sub 指令的引用，然后删除原有的 add 指令（eraseFromParent()）。
3. 完成对 handler.cpp 的编辑后,还需要对 main.cpp 进行修改,将 module 写入到文件中，具体实现可参照 llvm-dis 中的相关内容。

题目二思路：

1. 与题目一相似，首先需要检索出每一个 add 指令。
2. 若当前指令为 add 指令，追踪到跟随在其后面的 store 指令。对当前 block 进行分割(splitBasicBlock())，store 及其之前的内容作为原来的块（original block)，store 后的内容作为剩余的块(rest block)。
3. 构建一个新的块 sub100(BasicBlock::Create()),该块对 add 指令的值进行自减 100，并将自减后的数存储到原来 store 中的目的地址，最后跳转回到 rest block(BranchInst::Create())。
4. 在 store 所处的块中加入 branch 指令，若 add 指令的值大于 100，则跳转到 sub100，否则跳转到 rest block。

题目三思路：

1. 构建带有一个形参的函数（getOrInsertFunction()）。
2. 依次构建 3 个块（base、yes、no）,在 base 块中加入 branch 指令，当形参大于 100 时跳入到 yes,否则跳入到 no。
3. 在 yes 中对形参进行自减 100，并返回自减后的值（ReturnInst::Create()）。
4. 在 no 中直接返回形参值。
5. 检索除自建函数外的所有函数的 add 指令，追踪跟随其的 store 指令，在 store 指令之

前插入对自建函数的调用（CallInst::Create()），并修改原来的 store 的存储数据为调用后的数值。

题目四思路：

方法一：

1. 整体思路：将 c-fft 源代码转换为.ll 文件后，编写一个程序，使其在每个.ll 文件中插入 enter 函数和 exit 函数的声明（不需要具体定义），并且在每个 function（除 declaration 外）的第一行代码之前插入 enter 函数的调用，在每个 function 的 return 指令前插入 exit 函数的调用。
2. 参考 EHTest 中的相关例程，程序在每个.ll 文件中插入 enter 函数和 exit 函数的声明。具体例程：FuncEnterInfo（完成 enter 函数的声明）、FuncExitInfo（完成 exit 函数的声明）、CreateWords（创建一个字符串全局变量、并返回其指针）、CreateFuncEnterExit（基本逻辑：在函数的第一条指令之前插入 enter 函数的调用，参数为当前函数名，随后检测每一条指令，若其操作类型为 Instruction::Ret，在该指令之前插入 exit 函数的调用，参数为当前函数名）。需要注意字符串参数的创建和传递。
3. enter 函数和 exit 函数定义在另一个 Myfunc.c 文件中，enter 函数对当前的调用深度加一，并打印传递过来的字符串（即函数名）到一个文件中，exit 函数对当前调用深度减一。
4. 对 Myfun.c 编译为一个静态库 libmyfunc.a，完成插桩后的各个.ll 文件编译为.o 目标文件，最后一同链接，最终形成可执行文件 c-fft。c-fft 会在文件中记录函数调用关系图。

方法二：

整体思路同方法一，创建字符串全局变量时用 IRBuilder 的 CreateGlobalStringPtr 实现

题目五思路：

1. 该题与题目四相似，不同点在于需要处理多线程。多线程情况下，每个线程会并发地调用 enter 和 exit 函数，因此会使得打印混乱。需要解决该问题，需要将不同线程的函数调用图打印在不同文件中，并且，需要设置互斥锁或信号量，保护 enter 和 exit 函数中的全局静态数据结构，使其不发生数据竞争。因此，可保持题目四的其他文件不变，只需要改变 MyFunc.c 即可。
2. 需要构建两个重要的全局静态变量。第一个是 Map 结构数组，该数组的每个结构包含 4 个重要的成员，分别为当前线程 ID threadID、当前线程函数调用深度 depth、该结构是否被使用的标志 used、当前线程使用的文件 file。第二个是信号量 myfunc_mutex，用以保护 enter 和 exit 函数，确保其不被并发地执行。
3. 对于 enter 函数：在 main 函数被调用时，初始化 Map 结构数组（depth=-1,used=0）及信号量（信号量值初始化为 1，确保只有一个线程能够进行 enter 或 exit），该初始化过程只进行一次。
4. 随后使用信号量 myfunc_mutex 进行加锁。随后获取当前线程 ID，判断该线程 ID 是否已经储存在 Map 结构数组中，若存在，得出其数组索引；若不存在，说明该线程是一个新的线程，寻找数组中一个未被使用的结构，存放其线程 ID，并将 used 设置为 1，并得

出其数组索引。

5. 将对应数组索引的结构中 depth 自加 1。
6. 随后打开文件，将当前线程的函数调用深度、及传递过来的函数名打印到对应文件 file 中。完成该步骤后对信号量 myfunc_mutex 进行解锁。
7. 对于 exit 函数：首先对信号量 myfunc_mutex 进行加锁，获取当前线程 ID，并根据线程 ID 寻找对应的存放在 Map 数组中的结构，对该结构中的 depth 成员自减 1。
8. 若此时 depth 成员为-1，说明该线程的函数调用栈为空，线程即将结束，将该结构的 used 设置为 0。完成该步骤后对信号量 myfunc_mutx 进行解锁。

题目六思路：

1. 先把不同的.ll 文件读入，转换为 Module，然后构建一个 map，储存所有 Module 的函数名和 Function*的对应关系，跳过函数声明。
2. 声明一个 stack 储存指令块起止，stack 的每个元素是一个结构体，结构体储存了该指令块的起始和结束，声明一个 vector 储存路径上已经存在的 BasicBlock*，用以检测环的存在。
3. 声明一个函数 BBNode，该函数接受 2 中描述 stack，vector 数据和 BasicBlock*，以及深度信息，如果 BasicBlock*在 vector 中，出现环，返回。否则，把 BasicBlock*存入 vector。打印 BasicBlock*信息。深度加 1。
4. 如果栈不空，弹栈，遍历弹出元素表示的指令块
5. 遇到 Br 指令，按照跳转到的 BasicBlock 块数 n 拷贝 stack，把跳转到的 BasicBlock 的起始、结束信息分别入栈，分别递归调用 n 次 BBNode。返回；
6. 遇到 Call 指令，如果调用函数不在 map 中，略过，否则，把未遍历的指令起始、结束信息入栈，把调用函数的第一个 BasicBlock 的起始、结束信息入栈，递归调用 BBNode。返回；
7. 遇到 Ret 指令，把 vector 中属于该函数的 BasicBlock*删除

题目七思路：

算法背景请参照《Advanced COMPILER DESIGN IMPLEMENTATION》第 19.1 节。其核心思想为将函数视为节点，将调用语句视为边，通过迭代生成静态函数调用图。实现方法有很多，例如图的深度遍历/广度遍历，以及书上 19.1 节中给出的算法。

现将书上 19.1 节所述算法的伪代码呈现如下。

```
1.  E = set() # the set of edges
2.  oldN = set() # the set of processed nodes
3.  N = {rootFunction} # the set of visible nodes
4.  while not oldN.equal(N):
5.    func = N.substract(oldN).selectOne() # substract is difference operation o
    f set, selectOne pick one arbitrary element from set
6.    oldN.insert(func)
7.    for inst in func:
8.      for calledFunc in inst.callset:
9.        N.insert(calledFunc)
10.       E.insert(Edge(func, calledFunc, inst))
```

实现过程中需要注意以下几点：

1. 在静态函数调用图中，节点为函数，边为从调用者指向被调用者的有向边。在 llvm 中，指令不会调用同一个函数两次，所以可以将边建模为<调用函数，被调用函数，指令>的三元组。显然，静态函数调用图是多重图，或可表述为静态函数调用图的边是多标签的。
2. 每次 while 循环处理一个函数中的全部指令，E 为有向边的集合，oldN 为已处理函数集合，N 为与 E 有关的所有函数的集合。该算法如此设计的目的在于避免处理程序执行过程中不会被执行到的函数。显然，该算法没有考虑到函数指针等情况。
3. 处理函数调用语句时，将被调用函数插入 N 集合中，而将该语句对应的边插入到 E 集合中。

题目八思路：

建议阅读 Anderson 论文原文（Program Analysis and Specialization for the C Programming Language）。

建议按以下步骤实现：

1. **实现 intra-procedural 的简易版本**
2. **将第七题中实现的静态函数调用图引入**
3. **实现 inter-procedural 的复杂版本**

实现过程中需注意以下几点：

1. 在该论文的讨论中，变量被清晰地分为普通变量和指针变量两种。这两种变量都有与之对应的内存空间，而指针变量的值被认为指向某块内存空间。但是在 LLVM 中，其 IR 面向的是无线虚拟寄存器的虚拟机，源程序中所有变量都被映射为内存申请语句及引用其地址的寄存器。论文讨论语言和 LLVM 的差异导致语句处理方式的差异，需要建立论文讨论语言与 LLVM 之间的对应关系。

论文讨论语言语句	LLVM 语句	对应关系
a = b	%1 = load a Store %1, b	在源语言中的任何变量，在 LLVM IR 都会被映射为内存申请语句申请的内存。在进行约束限制时，限制的对象为虚拟寄存器指向的内存。
a = &b	%1 = alloc b %2 = alloc a store %1, %2	&b 的值即为%1,所以只需要把%1 存储到 a 对应的内存中。
a = *b	%1 = alloc b %2 = alloc a %3 = load %1 %4 = load %3 Store %4, %2	同上，%1 为指向 b 的指针，%3 为 b 的值，%4 为*b 的值。
*a = b	%1 = alloc b %2 = alloc a %3 = load %1 %4 = load %2	同上，%1 为指向 b 的指针，%2 为指向 a 的指针，%3 为 b 的值，%4 为 a 的值。最后的 store 将 b

	Store %3, %4	的值存储到a指向的内存中。

2. LLVM IR遵守SSA约束，虚拟寄存器只会被定义一次且会在使用前被定义。指针分析时这是一个非常好的特性，大幅度地降低算法实现的复杂性：维护从虚拟寄存器到图节点的关系非常简单。
3. Anderson算法是may-point-to算法，为此引入了Unknown机制，请在实现时慎重处理对Unknown地址的赋值。

题目九思路：

建议阅读Steensgaard论文原文（Points-to Analysis in Almost Linear Time)。

实现过程中需注意以下几点：

1. Steensgaard算法的主要思想为对每个变量建立对应的类型，在处理赋值等语句时改变变量之间的关系或对变量进行合并，从而实现`may point to`分析。该算法能达到O(n alpha(n))的时间复杂度和O(n)的空间复杂度，其中n指被分析程序的行数。该算法的时间和空间复杂度都比 Anderson 算法低，但是它分析的精确度也比Anderson算法低。
2. 同Anderson算法，在Steensgaard算法的实现中同样存在C语言与LLVM IR的差异导致的实现困难。具体来说，在Steensgaard算法中对变量的类型建模包含一般内存变量和 lambda 变量两种类型，该建模方式对LLVM IR中虚拟寄存器这种没有对应内存空间且只初始化一次的变量并不十分合适。所以推荐在实现时，仅将虚拟寄存器变量视为真实变量的标记，并将Steensgaard的类型建模到真实变量对应的内存空间。
3. 不同于Anderson算法，Steensgaard算法的分析结果中并没有对NULL和Unknown的考虑，也即在分析结果中无法分辨出对NULL或Unknown解引用的危险操作。
4. 论文中对算法的具体过程及数据结构的描述十分模糊，推荐在实现时参考其它博客等资料。

题目十思路：

1. 插桩的实现思路：遍历每个基本块，在其前插入一个函数调用。在函数调用内部使用共享内存，记录分支信息。插桩方法请参考题目三。分支记录方法可以在共享内存中存放一个hashmap，分支的唯一表示可以参考AFL，用上一个基本块的key与这个基本块的key异或后的值表示这个分支。
2. 将插桩应用他人项目：写一个 clang wrapper，在编译时设置环境变量如CC=/path/to/your_wrapper CXX=/path/to/your_wrapper++ ../configure 或者cmake .. -DCMAKE_C_COMPILER=/path/to/your_wrapper，可以在wrapper中开启debug选项并且添加’-fsanitize=address’编译选项开启ASAN
3. fuzzer 的种子变异等细节等可以参考 AFL(https://github.com/google/AFL)，honggfuzz(https://github.com/google/honggfuzz)。经典的种子变异方法有基本的比特反转，替换，拷贝，删除。
4. 运行目标程序并得到运行结果信息，其中实现细节是linux下多进程编程知识，如

fork(), execv(), waitpid(), pipe(), sigaction()等，标准的参考文档是 man page，也可谷歌找别人总结的文章。

5. Demo 代码里设置了五个类，“fuzzer”是入口，用其他四个类做组件进行 fuzzing。“Seedpool”维护所有种子的集合，负责种子排序与选取与删除。“Mutator”是函数指针，负责变异种子，其函数接受种子，输出一个新种子。“Runner”负责运行其目标程序。“Recorder”负责记录各种运行时信息包括分支覆盖率信息，并输出到屏幕。

参考资料：

1. llvm_doxygen-9.0 中各个类及其相关函数
2. llvm-9.0/examples 中的例程
3. https://releases.llvm.org/10.0.0/docs/index.html (llvm-9 的文档已经被删除)
4. DILP_llvm 中的例程
5. EHTest 中的相关例程
6. 《Advanced COMPILER DESIGN IMPLEMENTATION》
7. 谷歌