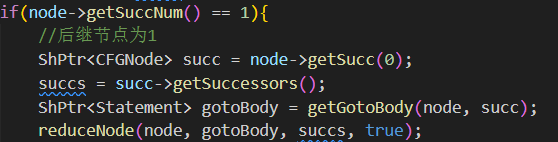
**A部分**

**任务一**

reduceCFG调用PoRECFGReducer::inspectCFGNode化简节点, 而reduceCFGComplete调用PoRECFGReducer::inspectCFGNodeComplete化简节点。这个将会在化简循环语句中使用到。

**任务二**

inspectCFGNodeComplete中的实现：



InspectCFGNode中的实现：



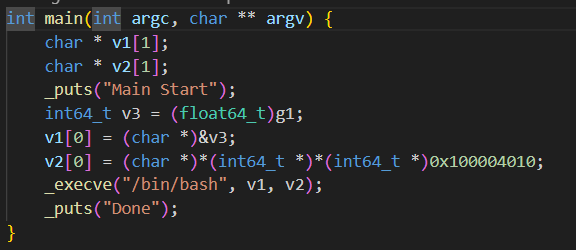
假设A、B两个节点为顺序结构，顺序结构的关键就是**保证A的后继节点只有一个B，并且B的前继节点只有一个A（否则会进入死循环）**。而且要排除自己指向自己的情况。

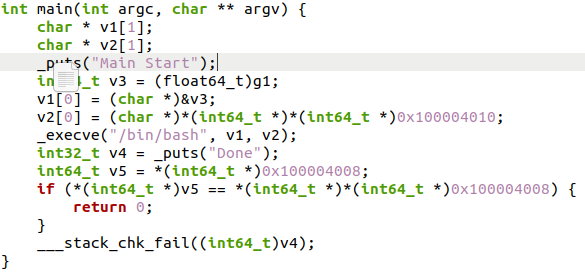
然后根据reduceNode的接口来获取succBody,nextSuccessors即可。

**两种规则对比**：

1. inspectCFGNode需要考虑的情况更多，否则会造成死循环
2. inspectCFGNodeComplete只需要考虑后续节点为一这一个条件

**完成任务前后不同sequen.bin.c对比：**

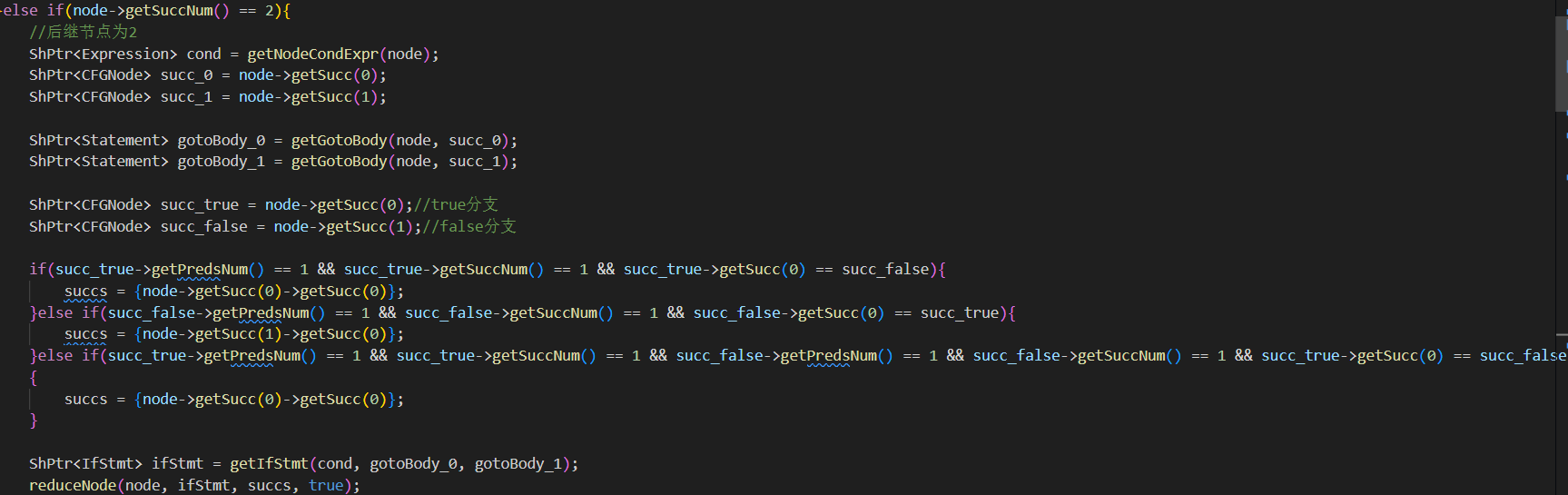
**前：**

**后：**

前后基本没差别

**任务三**

inspectCFGNodeComplete的实现：



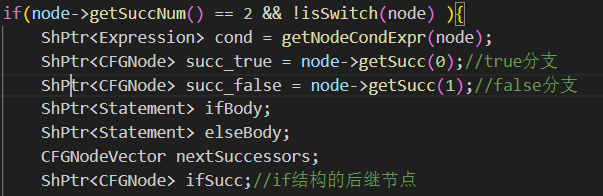
主要需要考虑的是当前节点的succs应该如何获取。

1. 如果是简单if语句，实现cond语句会跳转到true语句，那么下一节点就是node->getSucc(0)->getSucc(0)
2. 如果是简单if语句，实现cond语句会跳转到false语句，那么下一个节点就是node->getSucc(1)->getSucc(0)
3. 如果是if-else语句，那么下一节点直接就是node->getSucc(0)->getSucc(0)
4. If结构的识别与化简

首先是if结构的判断：

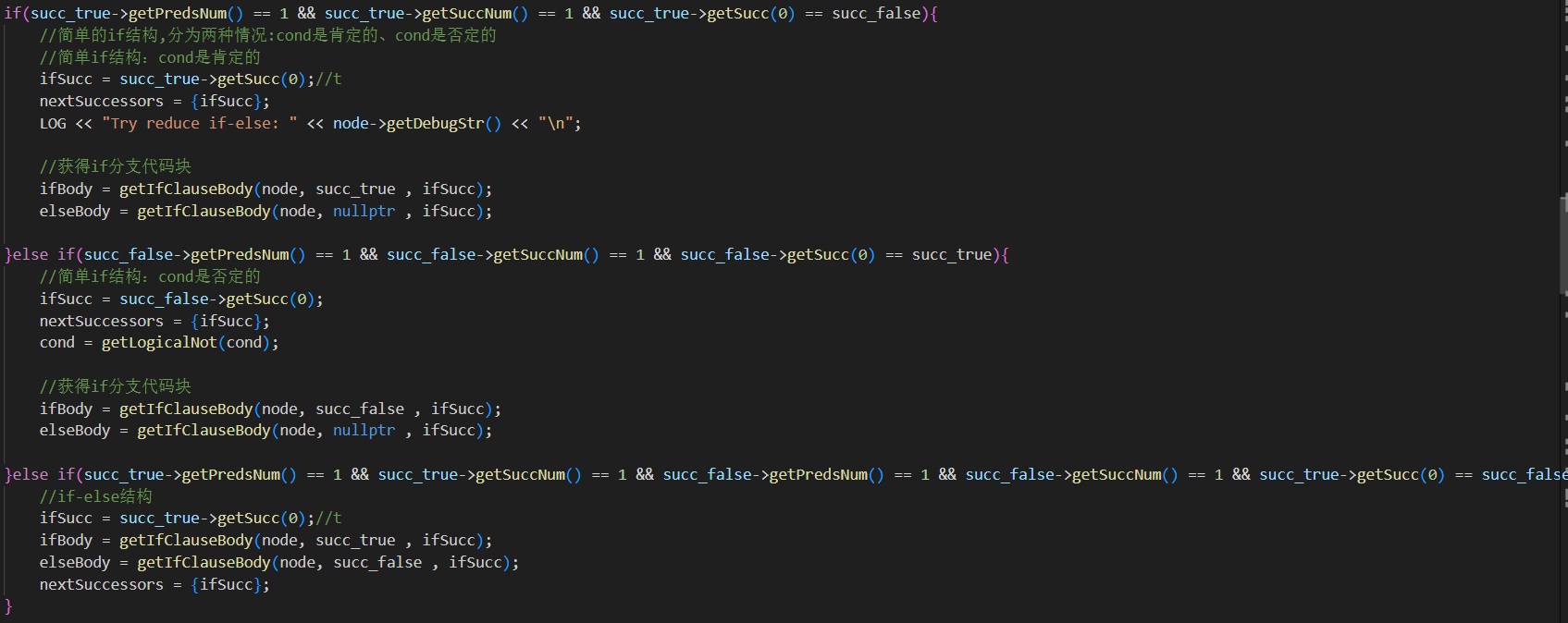
注意：判断条件不能有!isLoopHeader(node)，不然会识别不出来while-if的结构

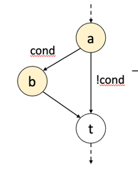
然后根据getIfClauseBody, getIfStmt的接口来确定需要获取的变量。

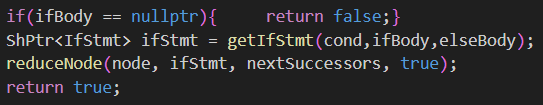


注意这里的ShPtr<Expression> cond = getNodeCondExpr(node);必须确定是if结构才能使用，否则会报错。

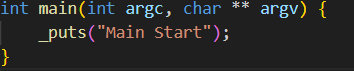
接下来需要判断三种情况，和上述inspectCFGNodeComplete的实现一样：

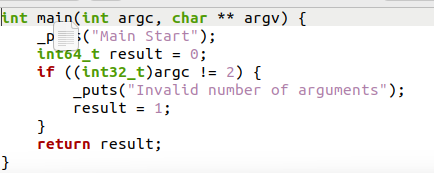


需要注意的是，如果是简单if结构，实现cond语句会跳转到true语句，那么elseBody的分支跳转节点是nullptr.根据这张图可以看出来原因是b和a都有共同的后继节点，不是if-else结构。

如果是简单if结构，实现cond语句会跳转到false语句，那么ifBody的分支跳转节点是nullptr。理由同上。注意同时还要有cond = getLogicalNot(cond);这是为了确保后面ifStmt的统一进行。

以branch\_if.bin.c为例，

前：

后：

其他包含if结构的也一眼能够正确识别if结构。

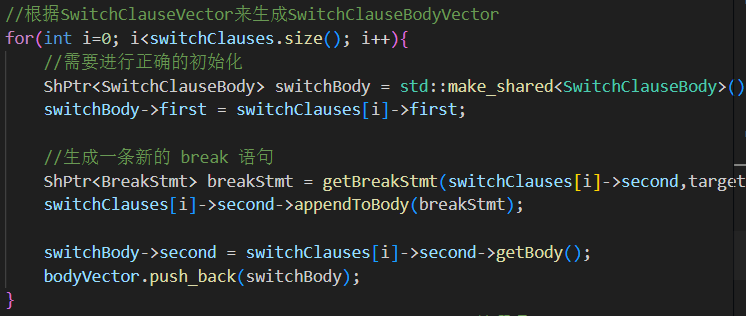
与Complete的对比：

这个方法需要考虑的情况更多，但是能够更加精确地识别if-else结构和简单的if结构。Complete需要考虑的情况更少。

1. Switch结构的识别与化简

遇到的主要困难是：对API的理解出现误差。误以为getSwitchStmt的第一个参数是SwitchClause = std::pair<ExprVector, ShPtr<CFGNode>>的第二个参数。

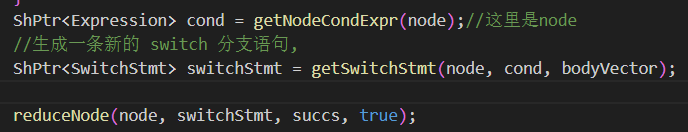
根据getSwitchStmt()的第三个接口const SwitchClauseBodyVector & clauses（switch 语句的所有分支）发现需要先循环一遍switch语句获得所有分支。



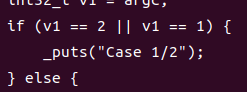
其中因为要加上break语句所以找了另一个API appendToBody。

需要注意的是，switchBody要先进行正确的初始化，否则后面->first会报错，空指向。

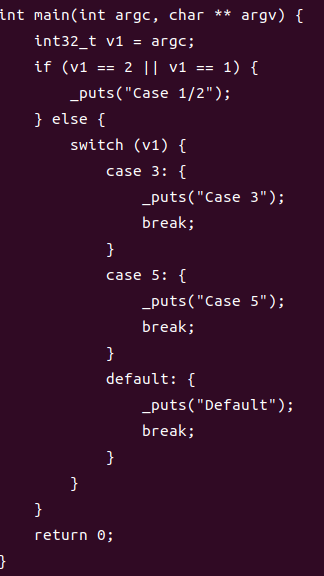
然后获得cond即可。



完成前：

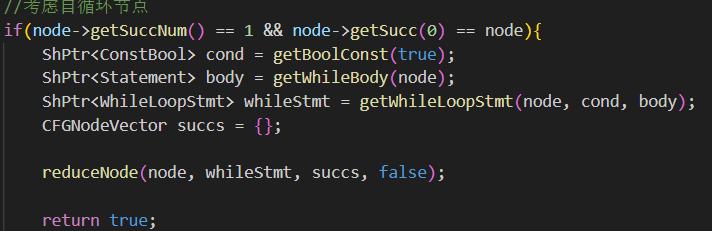


完成后：

与源代码语义相同。

**任务四：**

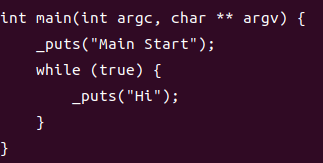
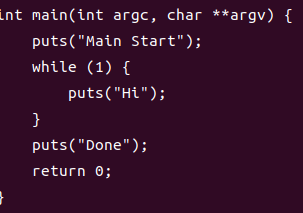
看到提示的getWhileLoopStmt()需要用到body，然后getWhileBody()要求是自循环节点，所以想到要先考虑自循环节点。



在这里遇到的困难：一开始没有正确定位自循环节点的条件node->getSuccNum() == 1导致一直报错。后来发现如果node->getSuccNum() > 1，那么会被后面的reduceCFGComplete(node)识别。

这样定位能够识别所有while(true)语句。但是无法识别while (i < argc)语句，只能通过goto来解决。

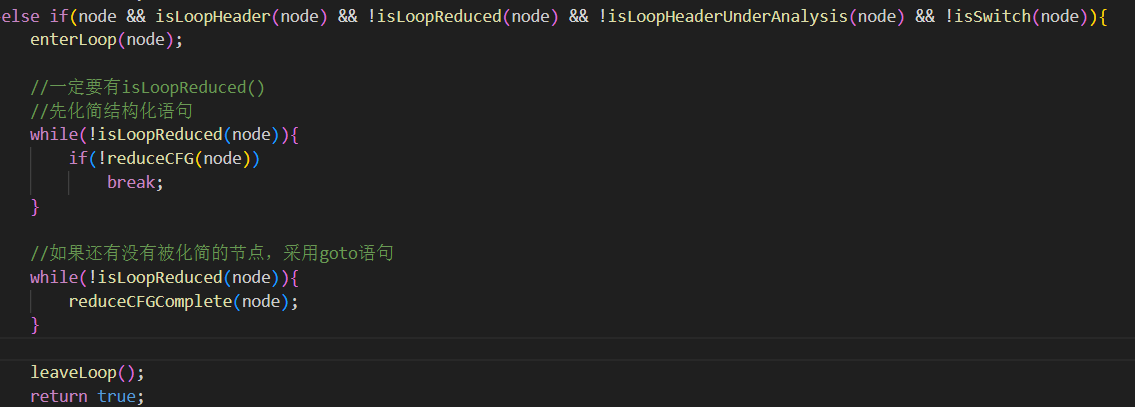
自循环节点不会跳转到其他节点，所以判断条件恒为true，并且也没有后续节点succs。

完成后与源代码语义相同。

再考虑非循环节点的情况。参考给出的提示。

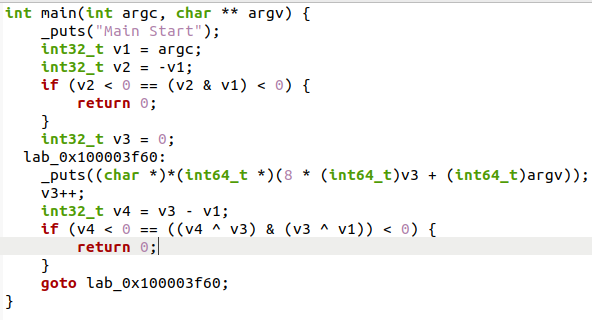
这里直接使用提示的API：reduceCFG, reduceCFGComplete

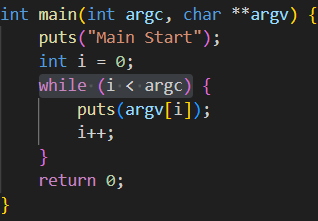
需要注意的是在使用之前都要确保这个循环没有被化简过，否则会报错。



reduceCFG调用PoRECFGReducer::inspectCFGNode化简节点, 而reduceCFGComplete调用PoRECFGReducer::inspectCFGNodeComplete化简节点。把while语句转化成多个if-goto语句的结合。

生成的结果：



与源代码语义相似

**B部分**

以这个片段为例：

============================函数定义============================

define i64\* @function\_100003d7c(i32 %size) {

dec\_label\_pc\_100003d7c:

  %0 = sext i32 %size to i64

  store i64 %0, i64\* @x0

; 0x100003d7c

  store volatile i64 4294983036, i64\* @\_asm\_program\_counter

  store i64 4294983680, i64\* @x16

; 0x100003d80

  store volatile i64 4294983040, i64\* @\_asm\_program\_counter

  %1 = load i64, i64\* @x16

  %2 = add i64 %1, 16

  %3 = inttoptr i64 %2 to i64\*

  %4 = load i64, i64\* %3

  store i64 %4, i64\* @x16

; 0x100003d84

  store volatile i64 4294983044, i64\* @\_asm\_program\_counter

  %5 = load i64, i64\* @x0

  %6 = trunc i64 %5 to i32

  %7 = call i64\* @\_malloc(i32 %6)

  %8 = ptrtoint i64\* %7 to i64

  store i64 %8, i64\* @x0

  %9 = ptrtoint i64\* %7 to i64

  store i64 %9, i64\* @x0

  %10 = load i64, i64\* @x0

  %11 = inttoptr i64 %10 to i64\*

  ret i64\* %11

}

============================定值使用============================

LLVM值Def:   %0 = sext i32 %size to i64

  被使用:   store i64 %0, i64\* @x0

LLVM值Def:   store i64 %0, i64\* @x0

  使用了:   %0 = sext i32 %size to i64

变量Def: 0x5b1faebe4088

  被使用:   %5 = load i64, i64\* @x0

LLVM值Def:

; 0x100003d7c

  store volatile i64 4294983036, i64\* @\_asm\_program\_counter

变量Def: 0x5b1faf125b28

LLVM值Def:   store i64 4294983680, i64\* @x16

变量Def: 0x5b1faebe5288

  被使用:   %1 = load i64, i64\* @x16

LLVM值Def:

; 0x100003d80

  store volatile i64 4294983040, i64\* @\_asm\_program\_counter

变量Def: 0x5b1faf125b28

LLVM值Def:   %1 = load i64, i64\* @x16

  被使用:   %2 = add i64 %1, 16

  使用了:   store i64 4294983680, i64\* @x16

LLVM值Def:   %2 = add i64 %1, 16

  被使用:   %3 = inttoptr i64 %2 to i64\*

  使用了:   %1 = load i64, i64\* @x16

LLVM值Def:   %3 = inttoptr i64 %2 to i64\*

  被使用:   %4 = load i64, i64\* %3

  使用了:   %2 = add i64 %1, 16

LLVM值Def:   %4 = load i64, i64\* %3

  被使用:   store i64 %4, i64\* @x16

  使用了:   %3 = inttoptr i64 %2 to i64\*

LLVM值Def:   store i64 %4, i64\* @x16

  使用了:   %4 = load i64, i64\* %3

变量Def: 0x5b1faebe5288

LLVM值Def:

; 0x100003d84

  store volatile i64 4294983044, i64\* @\_asm\_program\_counter

变量Def: 0x5b1faf125b28

LLVM值Def:   %5 = load i64, i64\* @x0

  被使用:   %6 = trunc i64 %5 to i32

  使用了:   store i64 %0, i64\* @x0

LLVM值Def:   %6 = trunc i64 %5 to i32

  被使用:   %7 = call i64\* @\_malloc(i32 %6)

  使用了:   %5 = load i64, i64\* @x0

LLVM值Def:   %7 = call i64\* @\_malloc(i32 %6)

  被使用:   %8 = ptrtoint i64\* %7 to i64

  被使用:   %9 = ptrtoint i64\* %7 to i64

  使用了:   %6 = trunc i64 %5 to i32

LLVM值Def:   %8 = ptrtoint i64\* %7 to i64

  被使用:   store i64 %8, i64\* @x0

  使用了:   %7 = call i64\* @\_malloc(i32 %6)

LLVM值Def:   store i64 %8, i64\* @x0

  使用了:   %8 = ptrtoint i64\* %7 to i64

变量Def: 0x5b1faebe4088

LLVM值Def:   %9 = ptrtoint i64\* %7 to i64

  被使用:   store i64 %9, i64\* @x0

  使用了:   %7 = call i64\* @\_malloc(i32 %6)

LLVM值Def:   store i64 %9, i64\* @x0

  使用了:   %9 = ptrtoint i64\* %7 to i64

变量Def: 0x5b1faebe4088

  被使用:   %10 = load i64, i64\* @x0

LLVM值Def:   %10 = load i64, i64\* @x0

  被使用:   %11 = inttoptr i64 %10 to i64\*

  使用了:   store i64 %9, i64\* @x0

LLVM值Def:   %11 = inttoptr i64 %10 to i64\*

  被使用:   ret i64\* %11

  使用了:   %10 = load i64, i64\* @x0

LLVM值Def:   ret i64\* %11

  使用了:   %11 = inttoptr i64 %10 to i64\*

其中，对于%0的情况，它通过"sext i32 %size to i64"这条指令定义，然后被使用在"store i64 %0, i64\* @x0"这个存储操作中，从而形成了DU链。

对于%5等变量，通过load指令加载数据到寄存器，然后进行一系列计算和转换操作，最终将结果存储回内存中，这些变量的定义和使用关系形成了DU链和UD链。

理解：

UD链：使用的变量在哪里被定义

UD链：定义的变量在哪里被使用

UD链：进行循环不变计算的检测，判定变量x是否未经定值就被引用，复制传播

DU链：无用代码的删除，循环中的代码外提

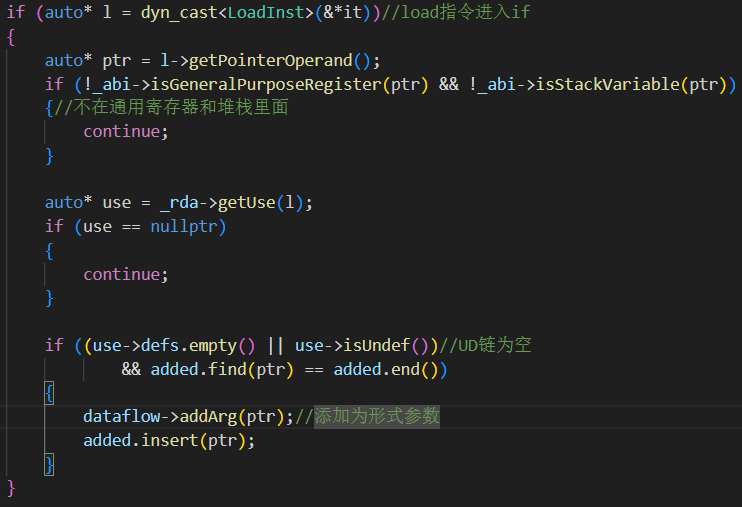
DU链和UD链的形成使得我们能够准确地定位变量的定义和使用位置，简化了数据流分析的复杂性。通过这些链，能够轻松追踪数据流，识别变量间的依赖关系，从而帮助化简数据流分析。

**任务二**

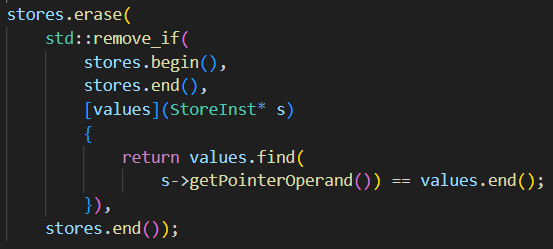
**2.2.1**

1. 插件具体是如何识别形式参数和实际参数的

(1)collectAllCalls调用createDataFlowEntry，createDataFlowEntry调用collectDefArgs，主要看collectDefArgs函数。



可以看到识别形式参数的主要方式是利用定值-使用分析，查找不在通用寄存器和堆栈变量中的load指令，分析函数中的未初始化引用。即：指令使用的操作数，UD链为空。

(2) collectAllCalls调用addDataFromCall，addDataFromCall调用collectCallArgs函数，通过collectStoresBeforeInstruction收集和识别调用指令前的store指令，并且能够过滤掉与上下文不相关的存储指令。

2. 在生成优化后代码时(applyToIr)，分别如何使用识别的形式参数和实际参数

（1）修改函数定义：使用IrModifier::modifyFunction()函数修改函数的定义。该函数会生成一个新的函数定义，并替换原有的函数定义。

（2）更新函数调用：通过调用de.setCalledValue(newFnc)更新函数调用指令中的被调用函数值，使其指向新生成的函数定义。

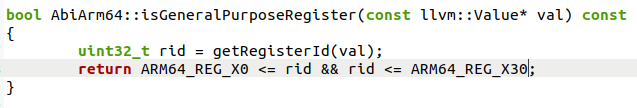
（3）构建参数列表和类型:根据收集到的形式参数信息（de.args()和de.argTypes()），构建函数的参数列表和类型列表。如果某个参数的类型未收集到，则使用ABI的默认类型作为该参数的类型。

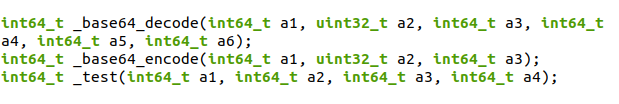
3.插件除了collectAllCalls外其它部分是怎么帮助优化函数类型识别和代码生成的

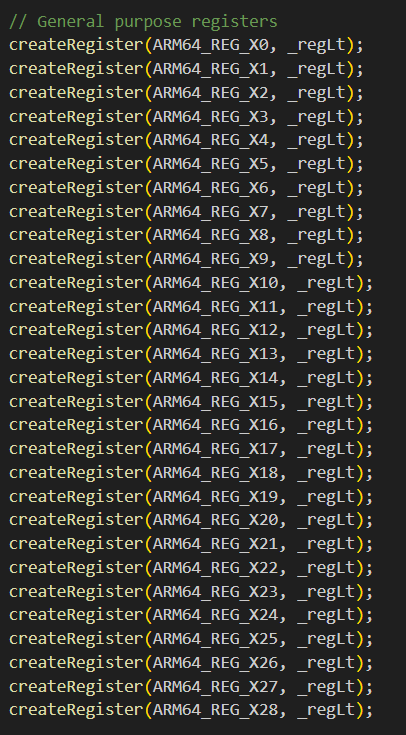
collectCallSpecificTypes函数分析调用约定。针对变参函数，尝试从函数调用中提取格式字符串（如printf风格的格式字符串），并使用这些信息来推断参数类型。这对于提高函数类型识别的准确性非常有帮助。

collectDefRets和collectCallRets函数分别收集函数定义中的return指令之后的load指令（对于返回值类型分析）和函数调用之后的load指令（对于返回值的使用分析）。这有助于确定函数的返回类型，以及如何在后续代码中使用这些返回值。

**2.2.2优化目标1**

根据提示修改etdec/src/bin2llvmir/providers/abi/arm64.cpp。关注到abi部分的if (!\_abi->isGeneralPurposeRegister(ptr) && !\_abi->isStackVariable(ptr))。其中检查的范围是X0-X30，但是根据ARM64架构下调用约定，函数实际上只使用X0到X7之间的寄存器传递参数。所以缩小范围能够更准确地识别参数。

把X30改成X7即可得到



在arm/arm64.cpp中，通用寄存器只有X0~X28，所以范围只需要保持在这之间即可，把X30改成X28也能够解决问题。

**2.2.3优化目标2**

分析.ll文件中byteswap32\_to\_host函数的调用者发现它自身并不接受任何参数。

* 1. **任务3：改进反编译器**

任务2.2.2中修复bug后，参数仍然存在问题，无法识别出来const