实验三：基于 Z3 和 LLVM 的程序分析

本次实验基于 Z3 和 LLVM 完成简单的使用 SMT 求解器的程序分析。

LLVM IR 简介与 LLVM C++ 接口的使用

LLVM IR

LLVM IR 是一种基于 SSA 的中间表示代码。对于（十分著名的）LLVM 这里不多做介绍，请尚不了解的同学求助于搜索引擎。LLVM IR 的 Reference Manual 见 https://llvm.org/docs/LangRef.html。

使用 Clang 可以从 C/C++ 等语言生成 LLVM IR 代码。例如，

```
clang -S -emit-llvm foo.c -o foo.ll
```

但 Clang 对于代码的处理并不非常的“SSA”。事实上，例如最简单的情形，

```
extern int a;
int foo(int i) {
    if (i == 3)
        return a;
    else
        return i;
}
```

生成的代码如下：

```
@a = external global i32, align 4

define i32 @foo(i32 %i) #0 {
entry:
  %retval = alloca i32, align 4
  %i.addr = alloca i32, align 4
  store i32 %i, i32* %i.addr, align 4
  %0 = load i32, i32* %i.addr, align 4
  %cmp = icmp eq i32 %0, 3
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %1 = load i32, i32* @a, align 4
  store i32 %1, i32* %retval, align 4
  br label %return

if.else:                                          ; preds = %entry
  %2 = load i32, i32* %i.addr, align 4
  store i32 %2, i32* %retval, align 4
  br label %return
```

```
return:                                           ; preds = %if.else, %if.then
  %3 = load i32, i32* %retval, align 4
  ret i32 %3
}
```

这里去除了一些无关内容。可以看出，它并没有使用 `phi` 节点来完成 `%1` 和 `%2` 间的选择，而是使用了对 `alloca` 分配得到的局部变量进行 `store` 和 `load` 来取得不同分支的结果，而这会极大地复杂化我们的分析。为了应对这种情况，**首先将刚刚生成的 `foo.ll` 中的各种 attribute 信息删除（到只保留上文内容）**，然后使用 LLVM 的优化 `mem2reg`：

```
opt -mem2reg -S foo.ll -o foo-opt.ll
```

而后，`foo-opt.ll` 的内容为：

```
@a = external global i32, align 4

define i32 @foo(i32 %i) {
entry:
  %cmp = icmp eq i32 %i, 3
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %0 = load i32, i32* @a, align 4
  br label %return

if.else:                                          ; preds = %entry
  br label %return

return:                                           ; preds = %if.else, %if.then
  %retval.0 = phi i32 [ %0, %if.then ], [ %i, %if.else ]
  ret i32 %retval.0
}
```

经过了将内存操作转变为寄存器操作的优化后，出现了 `phi` 指令。它的功能是根据运行时先前是从哪一个 BasicBlock 来到这里的选择使用哪一个值。以上述代码为例，它在先前是从 `if.then` 跳转至 `return` 时选择 `%0`，否则选择 `%i`。LLVM IR 要求 `phi` 指令只能是基本块的第一条指令，并且需要涵盖所有的可能前驱基本块。

LLVM 的 C++ 接口

LLVM 的大部分组件可以作为独立的依赖库使用。本次实验中，需要用到的主要是 IR 的读取、数据结构和遍历。读取 IR 的部分已经准备好代码，我们主要介绍其数据结构和遍历。

LLVM IR 中虽然呈现出过程式的表达，但读取到的数据结构有着一定的数据流特征：例如，`llvm::BinaryOperator::getOperand(unsigned int)` 的返回值具备类型 `llvm::Value *`，而它同时也是 `llvm::BinaryOperator` 的基类。在使用中需要适应这样的表达，习惯了之后很方便。

为了遍历程序，我们继承类 `llvm::InstVisitor`，处理我们关心的指令。具体的每个指令的类型可用的方法请参见相关的头文件和文档。

Z3

本次实验使用 Z3 的 C++ 接口完成。

`z3::expr` 是所有 Z3 中的表达式的基类。

实验内容

本次实验内容分三个阶段。

控制流敏感的单一过程分析：数组越界检查

这一阶段中，你需要对输入程序的每个函数分别进行分析，检查其中的数组越界。

我们省去循环展开的步骤；也即，我们假定输入的程序是无循环的。如此，我们只需处理一个有向无环图。

数组越界检查上我们也做一点简化：我们只分析包含 `inbounds` 选项的 `getelementptr` 指令。对于这些指令，LLVM 认为其结果是未定义的，或“poisoned”，可以认为已经发生了运行时错误。

本次实验中只处理整型数据。LLVM IR 中的整型均是有位长的，因此，我们不使用 Z3 中的 int，而是使用和 LLVM 中整型等长的 bit vector 来表示我们所有的类型。LLVM IR 中使用 `i1`，即位长为 1 的整型表示布尔值。我们跟随这一设计，通过 Z3 中的 `ite` 来将各种比较的结果布尔值转换为位长为 1 的 bit vector。

本次实验中只处理定长数组的越界检查。Clang 中使用形如 `[N x i32]*` 类型的指针来描述分配在栈上的定长数组。这极大地方便了分析，我们可以在 `getelementptr` 时得到要处理的数组的长度而无需关心其在何处分配。

对于分支的处理，我们不考虑基于 `load` / `store` 的数据传递，而只分析 `phi` 指令。同学们可以使用 Clang 配合 `opt -mem2reg` 来生成这样的测试代码。

本阶段实验中需要处理的指令包括：

- `add`, `sub`, `mul`, `shl`, `lshr`, `ashr`, `and`, `or`, `xor`, `icmp`, `zext`, `sext`

  对于这些指令，应当正确处理其逻辑，将其代表的命题加入 solver 中。例如，`%cmp = icmp ult i32 %i, 1024` 生成命题 `(= cmp (ite (bvult i #x00000400) #b1 #b0))`。这里无需担心 `%i` 是否已经被处理过，例如如果 `%i` 是输入的参数，我们关于它没有任何先验知识，那么 Z3 会在求解中考虑其整个可能取值范围。

- `br`

  分支是控制流敏感的分析的关键。这里不做太多的提示，但注意一点：不应让分析程序遍历所有的可能分支线路，因为如果有连续若干个无关的 if/else，会导致分析程序运行时间指数性的增加。

- `getelementptr`

  我们只对标记 inbounds 且来源元素类型（`llvm::GetElementPtrInst::getSourceElementType`）为 `[N x i32]` 的 GEP 进行处理。这种 GEP 指令形如

  ```
  %p = getelementptr inbounds [1024 x i32], [1024 x i32]* %a, i64 0, i64 %i
  ```

  其未越界的条件应当是 `%i ⩾ 0 && %i < 1024`。为避免 Z3 中类型不匹配问题，对于 index（此处为 `%i`），我们一律作 signed extend 至长为 64 的 bit vector，而后和 `ctx.bv_val(1024, 64)` 作比较。

  对这一指令，需要进行分析（check）。首先使用 `solver.push` 构造新的 scope，加入**能够抵达当前基本块的条件**和**发生越界的条件**作为新的 assertion，调用提供的 `checkAndReport` 执行分析并进行相应的输出，而后 `solver.pop` 恢复原本继续分析。这是为了防止未越界条件污染原本的已知信息。

- `phi`

对于这一指令，应当根据不同的来源选择其结果。善用 `implies`。

上下文敏感的多过程分析：数组越界检查

在 IPA 中，我们讲到有多种处理子过程的方式。我们采用基于摘要的方法。这是由于分析得到的输入和输出的相关命题集合天然地构成了一个摘要，表达为输入和输出间的关系。

在 Z3 中，提供了谓词逻辑/函数（`z3::func_decl`）的表示方法，用以表达一个函数。形式上来说，我们预先不知道每个函数的语义，而只知道其 return 值（也即函数值）满足的一系列 assertion；通过使用 `z3::solver` 求解这一系列 assertion，我们可以得到一个包含了这个函数表达的 `z3::model`。通过 `z3::model::eval` 接口对定义过的函数及其参数求值，来获得求解出的函数表达。（这里不使用 `func_interp` 是出于后续使用方便考虑。）

但为了能够正确地提取出其表达，函数中的每个“寄存器”将不再是 Z3 中的常量，而是也是关于函数参数的一个函数；也就是说，需要为每个 LLVM 寄存器构造一个 `z3::func_decl`，作为和参数相关的中间结果；并且在每个断言上对所有参数进行 `forall`。如果我们有以下函数：

```
define i32 @foo(i32 %i) {
  %ret = add i32 1, %i
  ret i32 %ret
}
```

以 SMT-LIB 表示，其分析后生成的逻辑程序应是

```
(declare-fun foo (Int) Int)
(declare-fun ret (Int) Int)
(assert (forall ((i Int)) (= (ret i) (+ 1 i))))
(assert (forall ((i Int)) (= (foo i) (ret i))))
```

或按 Z3 的 Python API，整个求解 `foo` 的过程为

```
from z3 import *
Z = IntSort()
ret = Function('ret', Z, Z)
foo = Function('foo', Z, Z)
i = Int('i')
solver = Solver()
solver.add(ForAll(i, ret(i) == 1 + i), ForAll(i, f(i) == ret(i)))
solver.check()
solver.model().eval(f(i))
```

本阶段实验中不考虑递归，但要考虑嵌套调用。

本阶段实验中额外需要处理的指令只有 `call`。只考虑参数为多个整型、返回值为一个整型的函数，且只考虑其返回值不依赖于访存结果的情形。

- Bottom up 阶段中，不处理 `getelementptr`，以函数表达每个寄存器，构建好每个函数的 assertions，全部构造好后联合求解得到完整的 model，`eval` 得到每个函数的表达，以备 Top down 阶段使用。
- Top down 阶段中，，在每个函数体内使用预先存储好的 assertions，附加（`z3::solver::add`）上已求解出的所有（被使用到的）函数表达，然后在 `getelementptr` 处和上一阶段相同地进行分析。

进一步完善

第三阶段不作为本实验的要求，而作为大实验的替代选项。进一步完善的方向包括：

1. 处理循环；
2. 处理递归；
3. 对内存建模，处理复杂的 `alloca`、`malloc`、`load` / `store` 等。

实验代码的编译运行和测试

本次实验代码使用 CMake 作为构建系统。

需要 LLVM 6.0.1 和 Z3 4.8.3，可能需要自行编译，均使用默认选项即可。稍后助教会在测试机上提供预编译版本以供使用。

为编译该程序，如下操作

```
mkdir build && cd build
cmake .. -DZ3_DIR="<Z3-installation>" -DLLVM_DIR="<LLVM-installation>/lib/cmake/llvm"
cmake --build .
```

为执行该程序，如下操作

```
./z3expr ../test/foo.ll
```