编译原理专题训练第三次实验

李晨昊 2017011466

2019-12-16

目录

1 实验环境

```
$ llvm-config --version
9.0.0
$ z3 --version
Z3 version 4.8.7 - 64 bit
```

不保证低版本下能编译通过。

2 实验结果和思路

我完成了实验要求中的前两个阶段，即单一过程分析和跨过程分析。对于自带的测例 `foo.ll`，我的代码可以正确识别出前三条和最后一条 `getelementptr` 指令不可能越界，而其余 `getelementptr` 指令可能越界。关于最后一条 `getelementptr` 指令，还有一些值得解释的细节，在后续报告中有详细说明。

提交的文件中，`main.cpp` 是我的最终成果，`main.cpp.task1` 是我完成了第一阶段之后的中间成果。这个中间成果只能识别出前三条 `getelementptr` 指令不可能越界，而不能正确判断最后一条。

下面我简单讲解一下我的实现思路，重点强调一些实验指导中没有说明的地方。

2.1 单一过程分析

我不是很喜欢这个继承的 `InstVisitor`，把继承关系删掉了。由于 llvm 摆脱了 c++ 原生的对象系统，自己造了一套性能较高的对象系统，判断一个对象的具体类型成为了一个低开销的常规操作，这使得使用访问者模式并没有什么实质性的意义了，所以我基本上把全部代码实现到很少的几个函数里去了。这样写 c++ 还算挺舒服的，有一点 rust 的 `match` 的感觉了。

只考虑单一过程分析的话，只需要遍历所有函数 (分析完一个函数之后可以清空分析的结果)，对于每个函数按照拓扑序遍历其所有基本块。之所以要采用拓扑序，是因为需要在进入一个基本块前，收集到所有进入这个基本块时成立的 assertions。由于我们不考虑循环，因此基本块之间是没有环的，所以拓扑排序总是存在的。具体实现上，我采用的是零入度算法，即 bfs 拓扑排序。

在 z3 中：

- 运算指令表示成类似 `dst == l + r` 的形式
- 比较指令表示成类似 `(dst != 0) == (l < r)` 的形式 (在第一版的代码中我使用了 `bool` 类型来表示 `dst`，不过后续为了简单还是改成了 1 位的位向量类型)
- `sext`，`zext` 指令可以用 z3 的 `sext`，`zext`
- 分支指令，条件分支则向两个出口基本块的进入条件中分别添加 `and(`本基本块的进入条件`, cond == 0)` 和 `and(`本基本块的进入条件`, cond != 0)`，无条件分支处理类似
- `phi` 指令，遍历所有前驱基本块 `b` 和对应的值 `v`，添加 `implies(`从 `b` 进入本基本块的条件`, dst == v)`
- `getelementptr` 指令，调用 `push`/`pop`，在 `push` 后添加`!(idx >= 0 || idx < len)` 和本基本块的进入条件

2.2 跨过程分析

按照实验指导中说的，为了实现跨过程分析，需要把每个非参数的寄存器都建模成一个关于参数的函数，而不是一个常量，而且不能在分析完一个函数之后清空分析的结果。为了避免名字冲突，这些寄存器的对应的函数名字都加上前缀`本函数的名字 $`。

现在，总体的流程变成：

1. 遍历所有函数，构造出对应的函数和参数并存储起来
2. 遍历所有函数,按照拓扑序遍历一个函数的所有基本块,这个阶段不处理 `getelementptr`
3. 遍历所有函数，遍历一个函数的所有基本块 (不需要按照拓扑序)，这个阶段只处理 `getelementptr`

第二个阶段中，需要把之前的类似于 `dst == l + r` 的 assertion，修改成类似于 `forall args: dst(args) == l(args) + r(args)` 的 assertion。

第三个阶段中，仍然使用 z3 提供的 `push/pop`，但是在 `push` 后，不能把`!(idx >= 0 || idx < len)` 修改成 `forall args: !(idx(args) >= 0 || idx(args) < len(args))`，因为这里检测的是可能越界，而非一定越界，所以需要修改成 `exists args: !(idx(args) >= 0 || idx(args) < len(args))`。

实验指导中说本阶段唯一一条需要额外处理的指令是 `call`，这个说法是错的，其实还有一条 `ret` 指令需要考虑。如果第二阶段中遇到 `ret` 指令，证明进入本基本块的条件 imply 本函数的函数值等于这个 `ret` 的函数值。

然而，进行到这里仍然不能分析出来最后一个 `getelementptr` 指令不可能越界。考虑 `foo.ll` 中的相关代码对应的等价的 c 代码：

```c
int32_t arr[1024];

int32_t abs(int32_t i) {
  if (i < 0) {
    return -i;
  } else {
    return i;
  }
}

int32_t interproc(int32_t i) {
  int32_t call = abs(i);
  if (call < 1024) {
    int64_t idxprom = (int64_t) call;
    return arr[idxprom];
  } else {
    return -1;
  }
}
```

那么，容易发现，**不能分析出来最后一个 `getelementptr` 指令不可能越界是完全正确的**，因为它就是有可能越界。只需考虑 `interproc` 的输入参数为 `i = -2147483648` 即可。

这个时候假如是在类似 java 这样的语言中，分析已经可以停止了，jvm 将无法（完全）优化掉 `interproc` 中的越界检查。然而如果是 c/c++，则还有一条额外的规则：**有符号整数溢出是未定义行为**。编译器/分析器的作者可以假定程序中永远没有未定义行为（除非分析的目的就是为了找这个未定义行为）。而上面描述的 case 之所以会发生越界，是因为在计算 `abs(-2147483648)` 时发生了有符号整数溢出，如果在 z3 中描述一条整数运算的时候加上不

会发生有符号整数溢出的条件，则 z3 就可以证明 `interproc` 中不会发生越界。为了简单起见，我只处理了加和减的情形，它们可以用 `implies` 加上一些简单的比较关系来描述。

本质上来说这个处理并不是针对 c/c++ 而言的，而是 llvm ir 自身的属性。在 c/c++ 中，“有符号整数溢出是未定义行为” 是无条件成立的，而在 ir 中，需要显式指出对有符号整数溢出是按未定义行为处理还是按回绕处理。在 ir 中这用 `nsw`，即 No Signed Wrap 标记来表示。这一点在 rust 中体现的非常明显：rust 中规定有符号整数溢出按回绕处理，但是也提供按未定义行为处理的 intrinsic 函数，例如：

```
pub unsafe fn add1(x: i32, y: i32) -> i32 { std::intrinsics::unchecked_add(x, y) }

pub fn add2(x: i32, y: i32) -> i32 { x + y }
```

生成的 llvm ir 为 (有一定修改)：

```
define i32 @add1(i32 %x, i32 %y) {
  %0 = add nsw i32 %y, %x
  ret i32 %0
}

define i32 @add2(i32 %x, i32 %y) {
  %0 = add i32 %y, %x
  ret i32 %0
}
```

总之，在处理 ir 的时候，只有遇到了 `nsw` 标记，才能告诉 z3 本次运算不会溢出，否则还是按照 z3 默认的逻辑，按回绕处理溢出。