



Clang静态分析的理解与实践

G1: 范文 吴钰同 孙一鸣 刘逸菲

答辩人: 范文 吴钰同



目录

01 基础问题回答

02 Checker 思路简介

03 Checker 演示

04 Q&A

The background of the slide features a complex, abstract geometric pattern. It consists of a dense network of thin, light gray lines that intersect to form a series of overlapping, curved shapes resembling funnels or hourglasses. Scattered throughout this network are numerous small, colored dots in various colors including red, yellow, green, blue, and purple. The central text is positioned within the white space of the slide, between the two main funnel-like structures.

PART 01

基础问题回答

Question & Answer 2.1

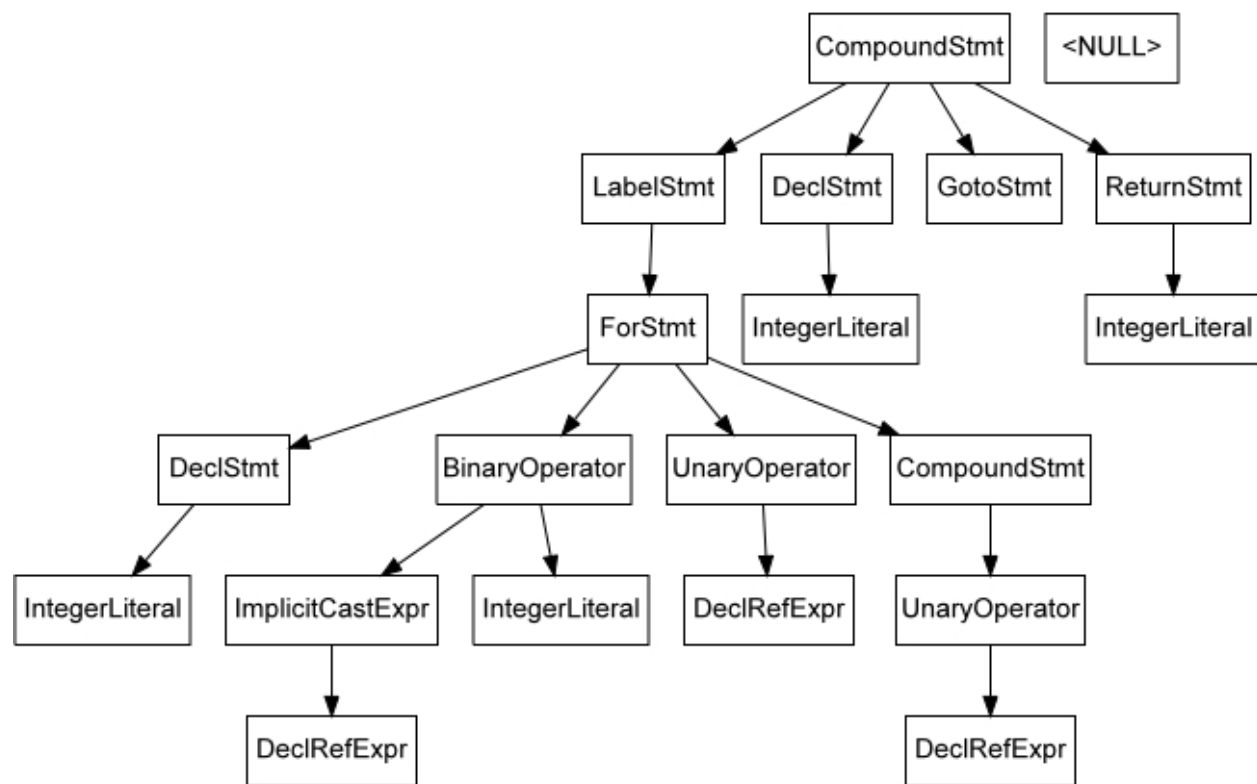
2.1

写一个含有循环、跳转逻辑的简单程序 test.c，绘制程序的AST、CFG 和 ExplodedGraph，简要说明区别和联系。

test.c:

```
int main() {  
    L1:  
    for(int i=0;i<2;i++){  
        i++;  
    }  
    int j = 0;  
    goto L1;  
    return 0;  
}
```

AST.svg:

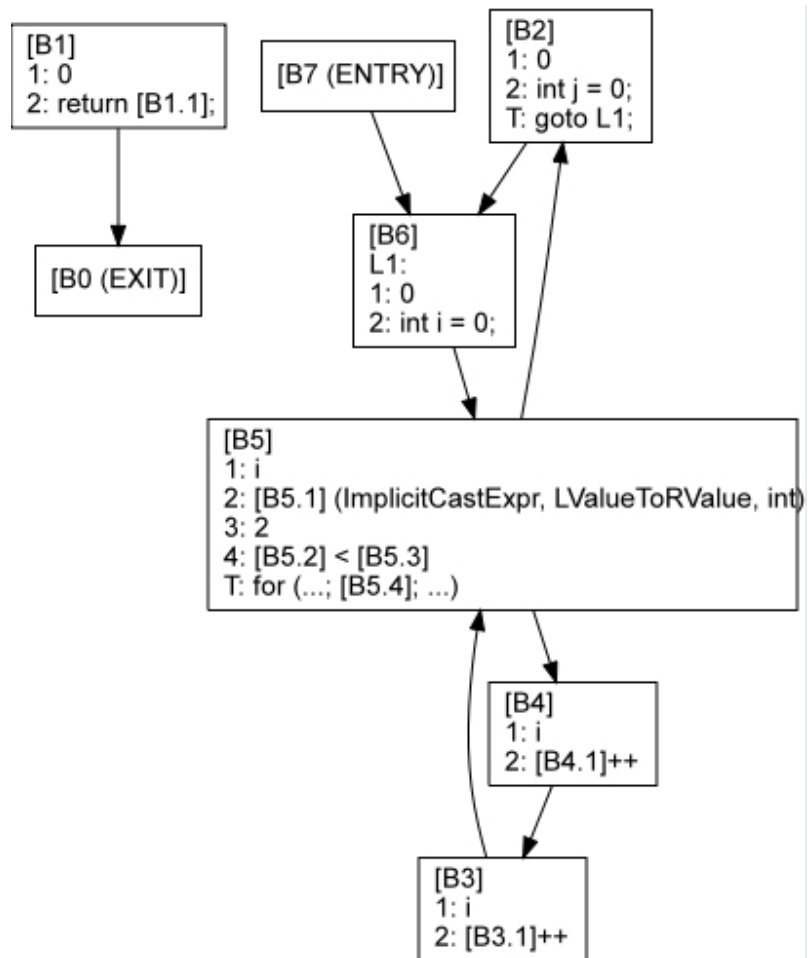


Question & Answer 2.1

2.1

写一个含有循环、跳转逻辑的简单程序 test.c，绘制程序的AST、CFG 和 ExplodedGraph，简要说明区别和联系。

CFG.svg:



Question & Answer 2.1

2.1

写一个含有循环、跳转逻辑的简单程序 test.c，绘制程序的AST、CFG 和 ExplodedGraph，简要说明区别和联系。

ExplodedGraph.svg:

```
{ "state_id": 238,
  "program_points": [
    { "kind": "Statement", "stmt_kind": "ImplicitCastExpr", "stmt_id": 595, "pointer": "0x55c526adff48", "cast_kind": "LValueToRValue", "pretty": "I", "location": { "line": 3, "column": 17, "file": "test.c" }, "stmt_point_kind": "P" },
    { "kind": "Statement", "stmt_kind": "ImplicitCastExpr", "stmt_id": 595, "pointer": "0x55c526adff48", "cast_kind": "LValueToRValue", "pretty": "I", "location": { "line": 3, "column": 17, "file": "test.c" }, "stmt_point_kind": "P" },
    { "kind": "Statement", "stmt_kind": "IntegerLiteral", "stmt_id": 591, "pointer": "0x55c526adff28", "pretty": "2", "location": { "line": 3, "column": 19, "file": "test.c" }, "stmt_point_kind": "P" },
  ],
  "program_state": {
    "store": { "pointer": "0x55c526b0c9ca", "items": [
      { "cluster": "I", "pointer": "0x55c526b0c930", "items": [
        { "kind": "Direct", "offset": 0, "value": "0 S32b" }
      ]
    }
  },
    "environment": { "pointer": "0x55c526b0c090", "items": [
      { "lctx_id": 1, "location_context": "#0 Call", "calling": "main", "location": null, "items": [
        { "stmt_id": 587, "pretty": "I", "value": "&i" },
        { "stmt_id": 595, "pretty": "I", "value": "0 S32b" }
      ]
    }
  },
    "constraints": null,
    "dynamic_types": null,
    "dynamic_casts": null,
    "constructing_objects": null,
    "checker_messages": null
  }
}
```

Question & Answer 2.2

2.2.1

Checker 对于程序的分析主要在 AST 上还是在 CFG 上进行?

CFG

2.2.2

Checker 在分析程序时需要记录程序状态, 这些状态一般保存在哪里?

在 ExplodedGraph 中的 ExplodeNode 里, 它包括 ProgramPoint 和 ProgramState, 即程序状态。

Question & Answer 2.3

2.3.2

LLVM 不使用 C++ 的运行时类型推断 (RTTI) , 理由是什么? LLVM 提供了怎样的机制来代替它?

C++ 的 `dynamic_cast<>` 运算符只能工作在有虚函数的类上, 并且其空间开销较大。使用如下模板类作替换:

`isa<>`: 类似 Java 中的 `instanceof` 运算符, 返回一个指针或引用是否是一个类型的实例, 返回 `True/False` 。

`cast<>`: 类型转换操作。它将指针或引用从基类转换为派生类, 如果不是正确类型的实例, 则会导致 `assertion failure` 。

`dyn_cast<>`: 检查转换操作。它检查操作数是否为指定的类型, 如果是, 则返回指向它的指针。如果操作数的类型不正确, 则返回空指针。

Question & Answer 2.4

2.4

SimpleStreamChecker

- 功能：识别文件重复关闭和文件指针变量生存期结束时文件未关闭导致资源泄漏的 bug。
- 测试程序

```
1 #include<stdio.h>
2 int main()
3 {
4     FILE *f = fopen("foo","r");
5     fclose(f);
6     fclose(f);
7     return 0;
8 }
```

- 文件重复关闭

```
../tests/test.cpp:6:5: warning: Closing a previously closed file stream [H2020.SimpleStreamChecker]
    fclose(f);
    ^~~~~~
1 warning generated.
```

```
1 #include<stdio.h>
2 int main()
3 {
4     FILE *f = fopen("foo","r");
5     return 0;
6 }
```

- 文件指针变量生存期结束时，文件未关闭

```
../tests/test.cpp:5:5: warning: Opened file is never closed; potential resource leak [H2020.SimpleStre
    return 0;
    ^~~~~~
1 warning generated.
```

Question & Answer 2.4

2.4.1

阅读 SimpleStreamChecker.cpp , 这个checker 对于什么对象保存了哪些状态? 保存在哪里?

- 保存文件流符号 (stream symbols) 的状态。
- 包括文件是打开还是关闭, 保存到 StreamState 类的 K 属性中。
- 还保存了符号索引 SymbolRef 到 StreamState 的映射, 保存到 StreamMap 中。

2.4.2

状态在哪些时候会发生变化?

- 在 checkPostCall 时如果发现有 fopen, 则在 StreamMap 中添加返回值 FileDesc 到一个类型为 Opened 的 StreamState 的映射。
- 在 checkPreCall 时如果发现有 fclose, 则在 StreamMap 中修改返回值 FileDesc 的映射对象为一个类型为 Closed 的 StreamState 。

Question & Answer 2.4

2.4.3 在哪些地方有对状态的检查？

- 在 checkPreCall 时如果发现有 fclose，则检查该 FileDesc 对应的 StreamState 是否已经被 close 了。
- 在 checkDeadSymbols 时检查该生存期已经结束的符号对应的文件是否关闭，即资源是否泄露。

2.4.4 函数 SimpleStreamChecker::checkPointerEscape 的逻辑是怎样的？实现了什么功能？用在什么地方？

遇到分析器未知的函数时，会自动调用 checkPointerEscape 。

- 逻辑：首先调用 guaranteedNotToCloseFile 判断函数调用 Call 是否会导致文件关闭，如果不会，则不需要继续处理。否则默认这些传递到未知函数的符号已经在某个地方被关闭了，将它们从 StreamMap 中删去。
- 实现了当被追踪的文件指针由于调用未知函数而 escape 的时候，不在子函数中继续追踪它。
- 用在调用分析器未知函数的时候，如只在本文件中声明而未在本文件中定义的函数。

Question & Answer 2.4

2.4.5 SimpleStreamChecker 有哪些局限性?

- 处理 pointer escape 时不再追踪已经打开的文件指针，而是默认它们关闭。
- 变量生存期结束时才检查是否有没有关闭的文件，无法处理全局变量重复赋值导致的资源泄露。

```
1  #include <stdio.h>
2
3  FILE *open(char *file)
4  {
5      return fopen(file, "r");
6  }
7
8  void f1(FILE *f);
9
10 int main()
11 {
12     FILE *f = open("foo");
13     f1(f);
14     return 0;
15 }
```

```
1  #include <stdio.h>
2  FILE* f;
3  FILE *open(char *file)
4  {
5      return fopen(file, "r");
6  }
7
8  int main()
9  {
10     f = open("foo");
11     f = open("bar");
12     fclose(f);
13     return 0;
14 }
```

Question & Answer 2.5

2.5.1

增加一个 checker 需要增加哪些文件？需要对哪些文件进行修改？

- 增加: 需要增加 checker 的 cpp 源代码文件，并对这个 checker 进行注册。
- 修改: 需要对 include/clang/StaticAnalyzer/Checkers/Checkers.td 进行修改，添加对应的 class 和 record。

2.5.2

解释 clang_tablegen 函数的作用。

通过阅读llvm官方文档可知，clang_tablegen 把目标描述文件(.td)翻译成 C++ 代码或者其他格式。

在 CMakeLists.txt 文件中，Checkers.inc 代表输出的结果文件，参数 -gen-clang-sa-checkers 指出用来生成 clang 静态分析器，SOURCE Checkers.td 指出输入为目标描述文件，TARGET ClangSACheckers 指出目标为clang静态分析器 ClangSACheckers。

Question & Answer 2.5

2.5.3

.td 文件的作用是什么？它是怎样生成 C++ 头文件或源文件的？这个机制有什么好处？

➤ .td 文件的作用：


通过阅读 llvm官方文档 可知，td(target descriptor) 中通过 abstract record 和 concrete record 描述了目标机器代码的简要信息。其中， abstract record 抽象了若干组 concrete record，使得 .td 文件更加简化。

➤ 如何生成 C++ 头文件或源文件：

调用 clang_tablegen 命令，解析 .td 文件中的 concrete record 并生成输出的 C++ 文件或者其他格式的文件。在此过程中，首先进行预处理，得到扩展的 class 和 record；然后通过作用域相关的后端 (domain specific backend)，得到 .inc 输出文件。

➤ 好处：

通过更简单的源文件，在生成复杂的 llvm 后端文件，因此可以减少重复代码,从而更加方便地构造作用域相关信息(domain specific information)。



PART 02

Checker 思路简介

Checker 思路简介



动态内存分配检查

1. malloc/new的实参是否为正数
2. 由于异常退出造成的malloc/new内存泄露
3. 动态内存分配的指针在strcpy中是否安全



析构函数检查

1. 检查析构函数中是否调用了exit或者throw
2. 检查析构函数中是否声明了static变量



静态变量检查

1. 检查静态局部变量的重复初始化

动态分配内存检查

1. malloc/new的实参是否为正数

```
void test() {  
    int *p;  
    int n1 = -1;  
    p = new int[n1]; // warn  
}
```

MallocOverflowSecurityChecker
NewArgChecker

思路：在CFG上调用
malloc/new处，检查实参
的符号值是否介于1到
 $2^{63} - 1$ 之间

如何知道当前函数为malloc/new:

在每个函数调用前，调用checkPreStmt；
或者直接在出现CXXAllocatorCall的时候进行检查

如何判断符号值:

通过ConstraintManager类提供的
assumeInclusiveRangeDual方法

动态分配内存检查

2.

由于异常退出造成的
malloc/new内存泄露

```
void f(int, int);
int g(int *);
int h() { throw 1; };

void test() {
    // It is possible that 'new int' is called first,
    // then 'h()', that throws an exception and eventually
    // 'g()' is never called.
    f(g(new int), h()); // warn: 'g()' may never be called.
}
```

LeakEvalOrderChecker

思路：检查每个函数的参数中是否有开辟新的内存空间的函数和异常退出的函数共存的情况

检查 malloc/new 方法同上

调用isNoReturn方法，并且通过遍历函数体的AST来判断这个函数中是否有throw语句

动态分配内存检查

3.

动态内存分配的指针在
strcpy中是否安全

```
void test() {  
    const char* s1 = "abc";  
    char *s2 = new char;  
    strcpy(s2, s1); // warn  
}
```

CStringChecker

思路：在malloc之后，记录指针指向内存空间的大小，调用strcpy时对它们进行比较。

类似于SimpleStreamChecker，使用一个全局的map记录指针的符号值与内存空间的一一映射关系。

在调用strcpy(dst, src)时，比较src的字符串长度和dst指向的空间大小

析构函数检查

1.

析构函数中是否调用了exit
或者throw

```
class A {  
    A() {}  
    ~A() { throw 1; } // warn  
};
```

DestructorChecker

DestructorThrowChecker

思路：首先判断当前状态点
是否在析构函数中；
若在析构函数中，则判断里
面的是否有exit调用以及
throw语句

当checker从CFG进入一个函数
的时候，利用CheckerContext
提供的上下文信息，判断是否
进入析构函数

类似LeakEvalOrderChecker，在析
构函数中判断是否有exit和throw语
句

静态变量检查

1. 静态局部变量的重复初始化


```
int test(int i) {  
    static int s = test(2 * i); // warn  
    return i + 1;  
}
```

StaticInitReenteredChecker

思路: 如果函数f中的一个静态局部变量s被函数g初始化, 则检查g是否会调用f


通过CheckBeginFunction和CheckEndFunction, 分析当前函数调用的函数, 建立一个函数调用关系图(g指向f当且仅当g直接调用f)

在之后针对s的checkPreStmt中, 用DFS检查g能否到达f



PART 03

Checker 演示



PART 04

Q&A



Thanks