Verifying Determinism in Sequential Programs

Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, Michael Ernst
University of Washington

ICSE 2021

USTC S4Plus阅读组会　彭浩然

程序不确定性带来的问题

- **难以测试**
  - 测试预言必须对所有可能性保证正确结果
  - 太严格的测试预言可能导致不稳定的测试（flaky tests）
- **难以比较两次运行结果**
  - 无法微调输入数据/程序以观察输出变化
  - 无法使用差分测试等技术
- **难以让用户信任运行结果**

程序的不确定性来源

- **不确定性集合类型**
  - 遍历哈希表
- **系统调用**
  - 时间日期
  - 文件系统
- **随机数生成器**
- **并行调度（本文不考虑）**

```
public List<TypeVariable> getTypeParameters() {
  Set<TypeVariable> parameters = new
          HashSet<>(super.getTypeParameters());
  …
  return new ArrayList<>(parameters);
}
```

```
public String toString(TypedClassOperation this) {
  return join(",", this.getTypeParameters());
}
```

来自Randoop的不确定性例子

传统运行时不确定性检测方法缺陷

- **需要实际运行**
  - 部署成本大
- **无法保证完备性**
  - 是否能检测到不确定性依赖于不确定性代码片段是否被运行

设计目标

- **提出程序中确定性属性的规范**
- **静态、可靠地验证确定性属性**
  - 无警告 -> 保证每次运行结果相同
- **当规范允许时，也允许程序中存在不确定性**
  - 例子：检测哈希表中是否存在元素的函数中对哈希表的使用

基于类型系统的设计

- 使用**类型限定符**表示确定性
- 使用**类型完备性规则**约束集合类型
- 使用对**多态性**的额外支持增加精确性

- 理论基础
  - 类型系统的表达力等价于静态分析
  - 实现为类型系统+自动类型推断，等价于实现一个静态分析

一般的类型限定符

- 缩小类型表示的范围
- 一般具有格的性质
- const volatile int < const int < int
- const volatile int < volatile int < int

本文设计的类型限定符

- **Det < OrderNonDet < NonDet**
- **NonDet**
  - 该对象两次执行中可能计算出不同的结果
- **OrderNonDet**
  - 该对象是集合或迭代器类型，每次执行包含相同元素，但给出元素的顺序两次执行可能不同
- **Det**
  - 该对象两次执行中永远计算出相同结果

集合类型完备性规则

- **完备性属性是一个深属性（deep property）**
  - 若集合/对象为NonDet，则其中取出的元素/域为NonDet
- **OrderNonDet集合**
  - OrderNonDet集合中取出的元素为NonDet
  - OrderNonDet集合的大小为Det
- **交汇取最小上界**
  - 赋值取左值（原）限定符和右值限定符的最小上界
  - 域访问取对象限定符和域限定符的最小上界

```
NonDet List⟨NonDet Int⟩          NonDet List⟨OrderNonDet Set⟩          NonDet List⟨Det Int⟩
~~OrderNonDet List⟨NonDet Int⟩~~   OrderNonDet List⟨OrderNonDet Set⟩   OrderNonDet List⟨Det Int⟩
~~Det List⟨NonDet Int⟩~~           ~~Det List⟨OrderNonDet Set⟩~~         Det List⟨Det Int⟩
```

多态支持

- **对于方法调用，多态地分析参数和返回值的限定符**
- **非多态的方法签名导致精度严重损失**
  - NonDet E get(NonDet List<E> this, NonDet int index)
  - NonDet E get(OrderNonDet List<E> this, Det int index)
  - Det E get(Det List<E> this, Det int index)
    - 很少情况所有List都是Det的、能一次NonDet都不出现然后分析得到Det签名
- **用多态签名捕捉尽可能多的Det情况**
  - PolyDet(up) E get(PolyDet List<E> this, PolyDet int index)
- **流敏感性**
  - 多态让类型限定符可以拥有流敏感性，每个程序点拥有不同限定符

实验结果

- **找到实际项目中的bug**
  - 其中Randoop声称非常注重确定性，在2017年7月花了整整两周专门找不确定性bug
- **与NonDex和DeFlaker的实证分析对比**
  - 找到所有它们举出的串行不确定性bug
  - 在本文的实证分析中额外找到13个bug

Project	LoC	Bugs
Randoop	23849	15
Checkstyle	36182	13
CF dataflow	13235	43
Plume lib	15220	16
Total	**88486**	**87**

实证分析

常见bug原因

- HashSet和HashMap的使用
- CLASSPATH环境变量的使用
- 比较器（Comparator）对等价元素的处理
- toString()方法打印顺序不同

常见误报原因

- 算法自身接受乱序输入
- 具体类的toString()实现被标为Det而其接口类未被标为Det
- 程序内实现了缓存结构，“缓存”内容不确定，但“主存”内容确定
- 集合排序可以让OrderNonDet变得有序，但无法识别

文章亮点

- **把静态分析实现为类型系统+自动类型推断**
  - 表达力等价于静态分析
  - 文章中描述更清晰
- **实证分析详细**
  - 统计常见bug的例子
  - 统计常见误报原因
  - 与其他文章的实证分析对比