软件筑基智能新时代
2026 CCF ChinaSoft 中国软件大会
NASAC 第25届全国软件与应用学术会议 The 25th National Software and Application Conference
FMAC 第11届全国形式化方法与应用会议 The 11th National Conference on Formal Methods and Applications

Datalog程序的自动缺陷定位

汇报人：钮鑫涛

汇报时间：2026年11月6日



Automated Debugging of Datalog Programs










Jiashen Wei* Baoyuan Luo* Runshuo Xie Yun Qi Yiyu Zhang Xizao Wang Xintao Niu# Zhiqiang Zuo#

*：同等贡献 #：通讯作者

Nanjing University, China

OOPSLA 2026 Distinguished Paper Award

CONTENTS | 目录

01 Datalog简介

02 证明树调试

03 新视角：面向 Datalog 的 SBFL

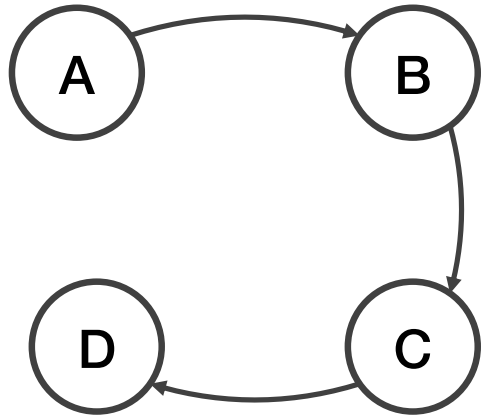
04 基准构建

05 实验评估与总结

Datalog 简介

- Datalog 是一种声明式编程语言
- Datalog 程序由规则（*rules*）组成，规则作用于输入事实（*facts*）
- 输入事实描述已知的信息：

```
edge(a,b), edge(b,c), edge(c,d)
```

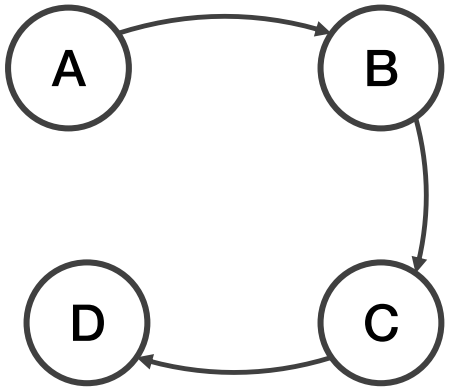


- 规则描述如何推导出新的事实：

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- path(X,Y), edge(Y,Z).
```

Datalog 简介

- 从概念上讲，规则被反复应用，直到达到不动点



r1
```
path(X,Y) :-
      edge(X,Y).
```

r2
```
path(X,Z) :-
      path(X,Y),
      edge(Y,Z).
```

当推导出的事实有错时，怎样找到有缺陷的规则?

edge(a,b).
edge(b,c).
edge(c,d).

→

edge(a,b).
edge(b,c).
edge(c,d).
path(a,b).
path(b,c).
path(c,d).

→

edge(a,b).
edge(b,c).
edge(c,d).
path(a,b).
path(b,c).
path(c,d).
path(a,c).
path(b,d).

→

edge(a,b).
edge(b,c).
edge(c,d).
path(a,b).
path(b,c).
path(c,d).
path(a,c).
path(b,d).
path(a,d).

可达性分析

```
class Main {
    static void main(String[] args) {
        init();

        Parser p = new JsonParser();
        p.parse();
    }

    static void init() {
        loadConfig();
    }

    static void loadConfig() {}
}
```

instantiate

```
interface Parser {
    void parse();
}
```

```
class JsonParser
implements Parser {
    public void parse() {
        readJson();
    }
    void readJson() {}
}
```

```
class YamlParser
implements Parser {
    public void parse() {
        readYaml();
    }
    void readYaml() {}
}
```

implements

可达性分析

被分析的 Java 程序

```
interface Parser {
    void parse();
}

class JsonParser
implements Parser {
    public void parse() {
        readJson();
    }
    void readJson() {}
}

class YamlParser
implements Parser {
    public void parse() {
        readYaml();
    }
    void readYaml() {}
}

class Main {
    static void main(String[] args) {
        init();

        Parser p = new JsonParser();
        p.parse();
    }

    static void init() {
        loadConfig();
    }

    static void loadConfig() {}
}
```

implements

instantiate

Datalog 分析器

r1

```
call_edge(M, N) :-
      direct_call(M, N).
```

r2

```
call_edge(M,N) :-
      virtual_call(M,T,S),
      method_impl(C,S,N),
      subtype(C,T).
```

r3

```
reachable(M) :-
      entry(M).
```

r4

```
reachable(N) :-
      reachable(M),
      call_edge(M,N).
```

推导出的事实

reachable(Main.main)

reachable(Main.init)

reachable(Main.loadConfig)

reachable(JsonParser.parse)

reachable(JsonParser.readJson)

reachable(YamlParser.parse)

reachable(YamlParser.readYaml)

可达性分析

被分析的 Java 程序

```
interface Parser {
    void parse();
}

class JsonParser
implements Parser {
    public void parse() {
        readJson();
    }
    void readJson() {}
}

class YamlParser
implements Parser {
    public void parse() {
        readYaml();
    }
    void readYaml() {}
}

class Main {
    static void main(String[] args) {
        init();

        Parser p = new JsonParser();
        p.parse();
    }

    static void init() {
        loadConfig();
    }

    static void loadConfig() {}
}
```

implements

instantiate

Datalog 分析器

r1

```
call_edge(M, N) :-
      direct_call(M, N).
```

r2

```
call_edge(M,N) :-
      virtual_call(M,T,S),
      method_impl(C,S,N),
      subtype(C,T).
```

r3

```
reachable(M) :-
      entry(M).
```

r4

```
reachable(N) :-
      reachable(M),
      call_edge(M,N).
```

推导出的事实

reachable(Main.main) ✓

reachable(Main.init) ✓

reachable(Main.loadConfig) ✓

reachable(JsonParser.parse) ✓

reachable(JsonParser.readJson) ✓

reachable(YamlParser.parse) ✗

reachable(YamlParser.readYaml) ✗

这两个错误事实是怎么来的?

CONTENTS | 目录

01 Datalog简介

02 证明树调试

03 新视角：面向 Datalog 的 SBFL

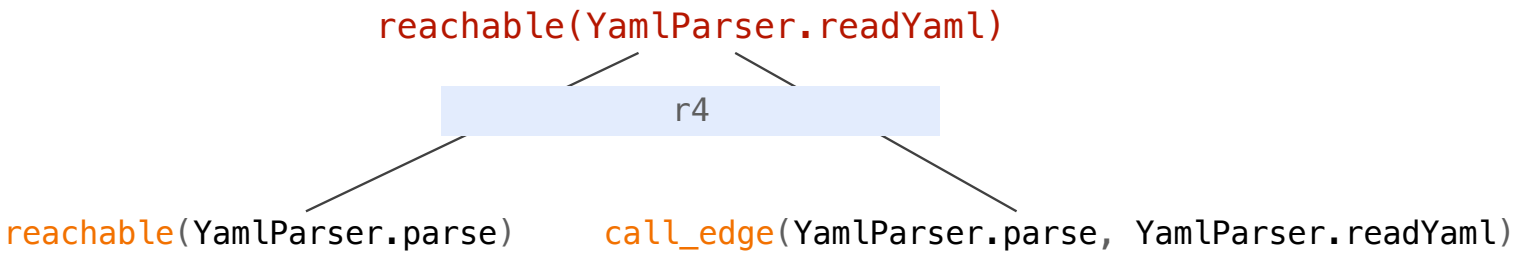
04 基准构建

05 实验评估与总结

用证明树调试

```
reachable(YamlParser.readYaml)
```

用证明树调试



r4

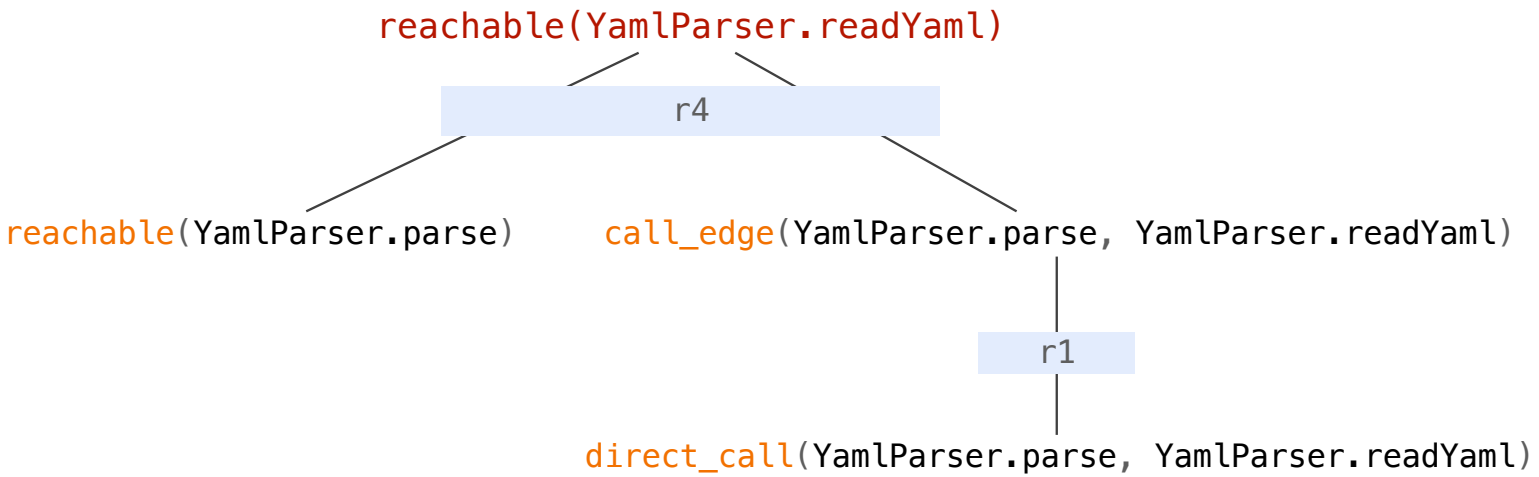
```
reachable(N) :-
    reachable(M),
    call_edge(M,N).
```

```
class YamlParser
implements Parser {
    public void parse() {
        readYaml();
    }
    void readYaml() {}
}
```

用证明树调试

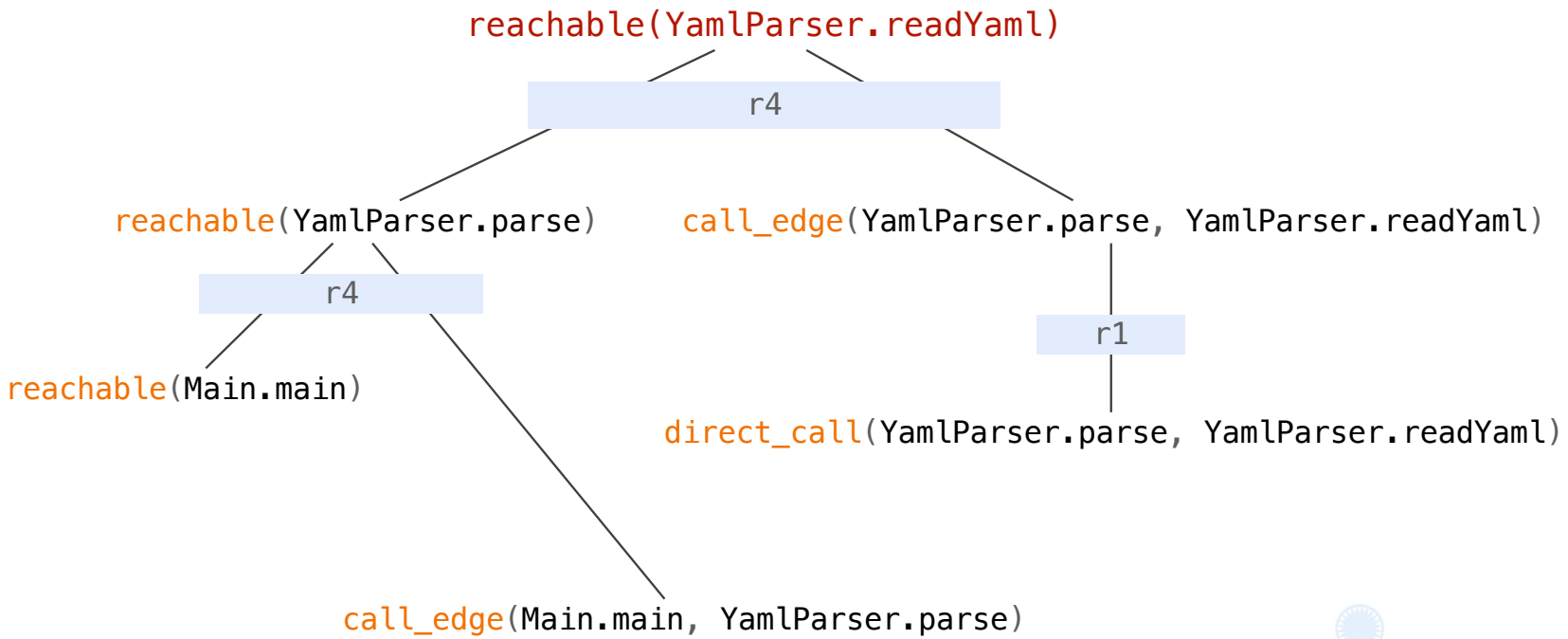
r1

```
call_edge(M, N) :-
      direct_call(M, N).
```




```
class YamlParser
implements Parser {
    public void parse() {
        readYaml();
    }
    void readYaml() {}
}
```

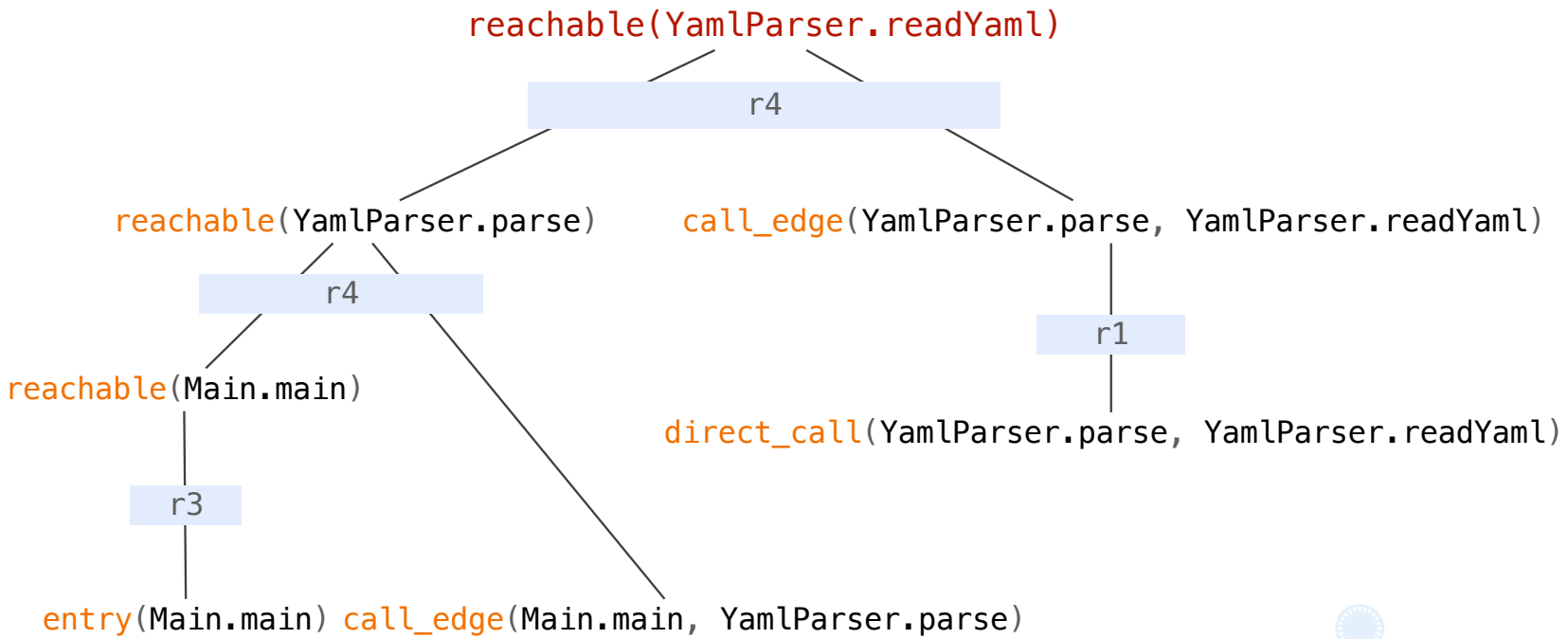
用证明树调试



r4

```
reachable(N) :-
    reachable(M),
    call_edge(M,N).
```

用证明树调试



r3

```
reachable(M) :-
      entry(M).
```

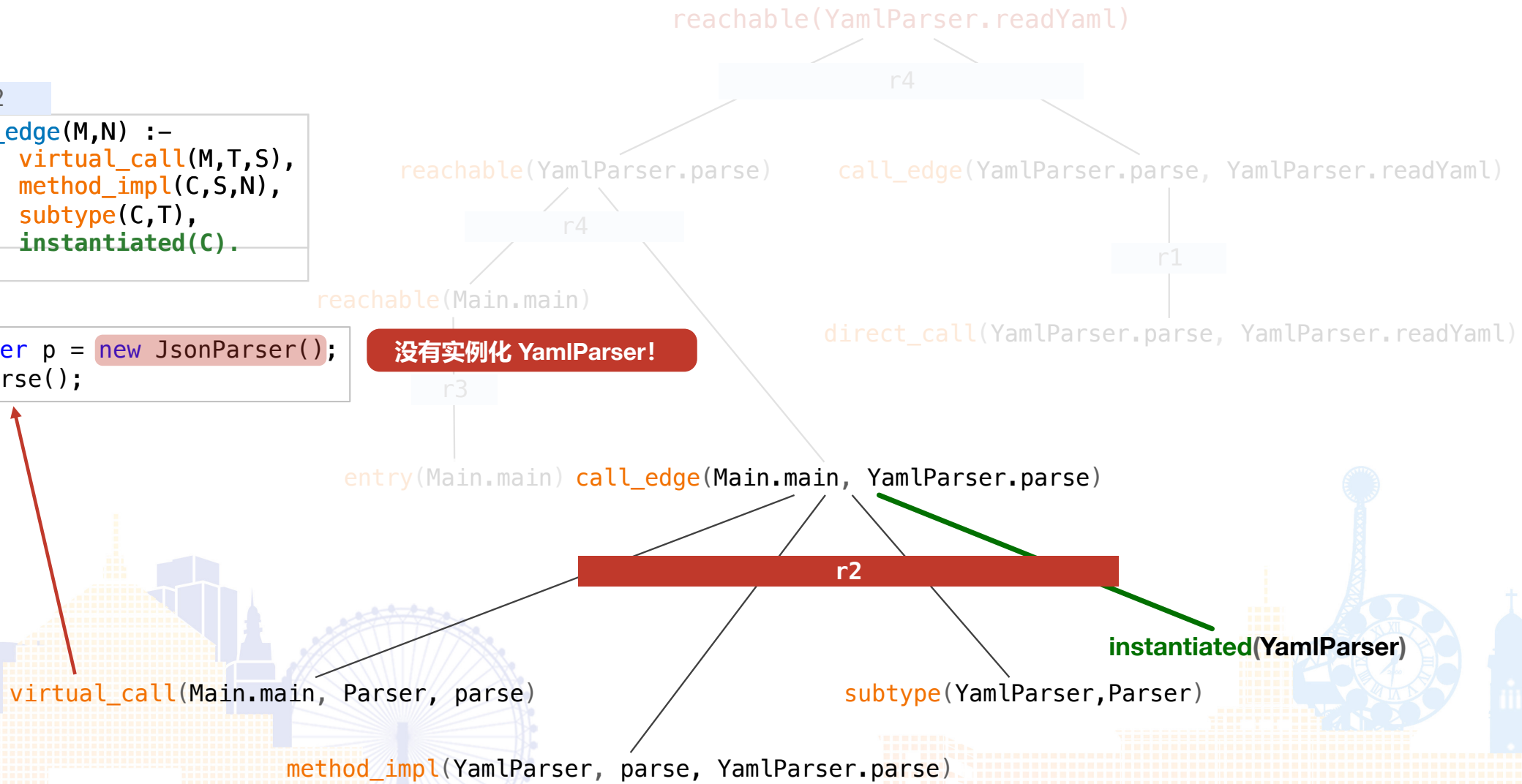
用证明树调试

r2

```
call_edge(M,N) :-
      virtual_call(M,T,S),
      method_impl(C,S,N),
      subtype(C,T),
      instantiated(C).
```

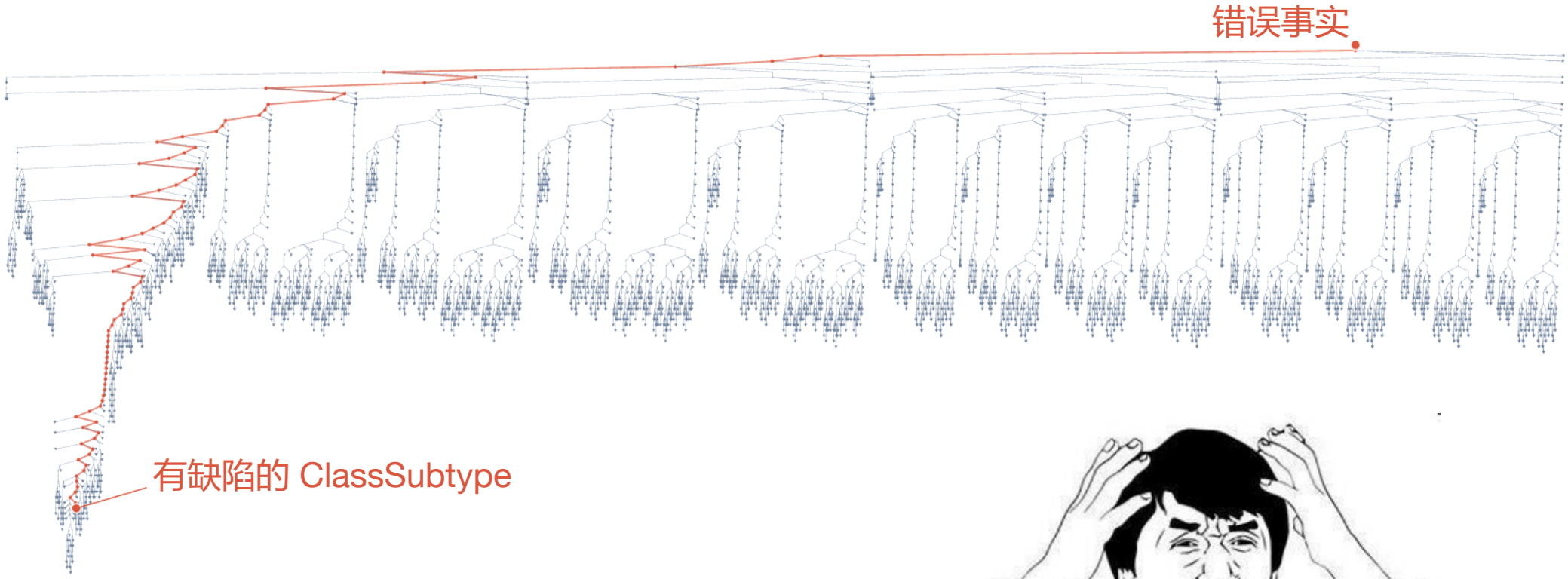
```
Parser p = new JsonParser();
p.parse();
```

没有实例化 YamlParser!



真实案例：Fault #31

- 调试一棵小的证明树还可以
- 那么大的呢？



- 29,018 个不同的事实
- 离最近的缺陷规则还有 86 条边

调试单个事实似乎非常困难

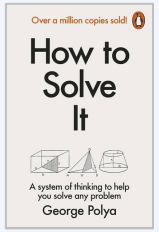
为什么不同时调试多个事实呢?



乔治·波利亚（George Pólya，1887–1985）

发明者悖论：更一般的问题，有时反而更容易解决

出自《怎样解题》（How to Solve It，1945）



例：费曼的积分技巧 把常数换成参数，问题反而变简单

1 原题：直接求很难

$$\int_0^1 \frac{x-1}{\ln x}\,dx = ?$$

被积函数没有初等原函数

→

2 放进一族，对 t 求导

$$I(t) = \int_0^1 \frac{x^t - 1}{\ln x}\,dx$$

$$I'(t) = \int_0^1 \frac{x^t \cancel{\ln x}}{\cancel{\ln x}}\,dx = \frac{1}{1+t}$$

→

3 积回去，原题只是 t = 1

$$I(0) = 0 \;\Rightarrow\; I(t) = \ln(1+t)$$

$$I(1) = \ln 2$$

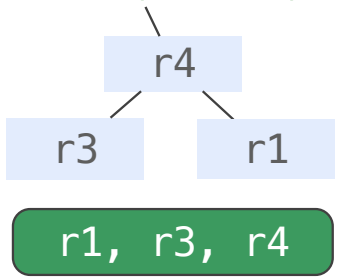
整族一次解出，原题顺带得到

CONTENTS | 目录

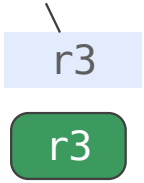
01 Datalog简介
02 证明树调试
03 新视角：面向 Datalog 的 SBFL
04 基准构建
05 实验评估与总结

新视角

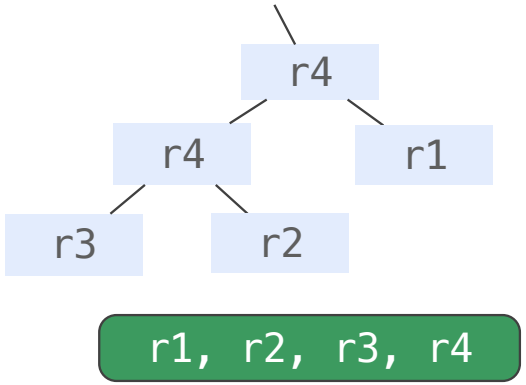
reachable(Main.init)



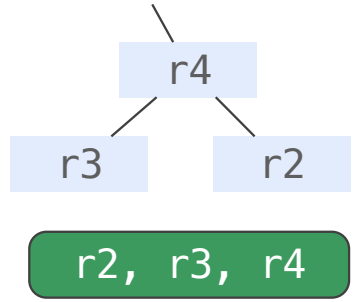
reachable(Main.main)



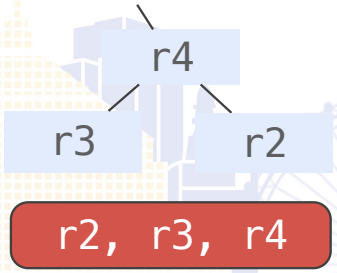
reachable(Main.loadConfig)



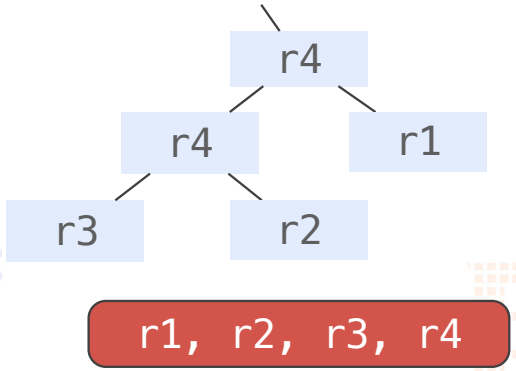
reachable(JsonParser.parse)



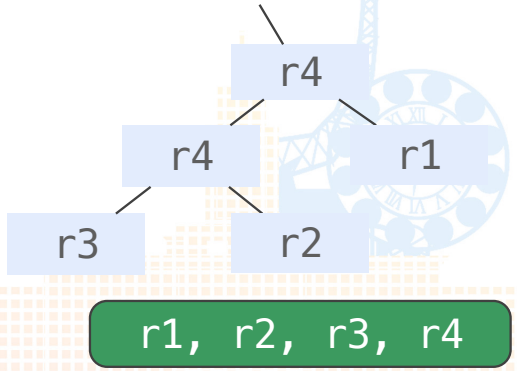
reachable(YamlParser.parse)



reachable(YamlParser.readYaml)



reachable(JsonParser.readJson)



新视角

输出事实	r1	r2	r3	r4
reachable(Main.main)			✓	
reachable(Main.init)	✓		✓	✓
reachable(Main.loadConfig)	✓		✓	✓
reachable(JsonParser.parse)		✓	✓	✓
reachable(JsonParser.readJson)	✓	✓	✓	✓
reachable(YamlParser.parse)		✓	✓	✓
reachable(YamlParser.readYaml)	✓	✓	✓	✓



约翰·斯诺（John Snow，1813–1858）

1854 年伦敦霍乱：单看一个病人，找不出病因

把病例和非病例放在一起对比 → 锁定宽街的水泵



斯诺 1854 年的霍乱地图（局部），黑色短条为死亡病例

新视角

输出事实	r1	r2	r3	r4
reachable(Main.main)			✓	
reachable(Main.init)	✓		✓	✓
reachable(Main.loadConfig)	✓		✓	✓
reachable(JsonParser.parse)		✓	✓	✓
reachable(JsonParser.readJson)	✓	✓	✓	✓
reachable(YamlParser.parse)		✓	✓	✓
reachable(YamlParser.readYaml)	✓	✓	✓	✓

- 有缺陷的规则往往更多地参与错误事实的推导，而较少参与正确事实的推导。

基于程序谱的缺陷定位

```
1.int calculatePrice(User user) {
2.     int price = 100;
3.     if (user.isPremium())
4.          price -= 10;
5.     if (user.hasCoupon())
6.          price -= 20;
7.     if (user.isStudent()) {
8.          price += 30;   // should be -= 30
9.   }
10.     return price;
11.}
```

被测程序

测试

✓ t_1

In: Normal

Expected: 100

Out: 100

```
1.int calculatePrice(User user) {
2.    int price = 100;
3.    if (user.isPremium())
4.        price -= 10;
5.    if (user.hasCoupon())
6.        price -= 20;
7.    if (user.isStudent()) {
8.        price += 30;   // should be -= 30
9.    }
10.    return price;
11.}
```

L2

被测程序

```
1.int calculatePrice(User user) {
2.    int price = 100;
3.    if (user.isPremium())
4.        price -= 10;
5.    if (user.hasCoupon())
6.        price -= 20;
7.    if (user.isStudent()) {
8.        price += 30;   // should be -= 30
9.    }
10.    return price;
11.}
```

L2

L4

测试

✓ t_2

In: Premium

Expected: 90

Out: 90

被测程序

```
1.int calculatePrice(User user) {
2.    int price = 100;
3.    if (user.isPremium())
4.        price -= 10;
5.    if (user.hasCoupon())
6.        price -= 20;
7.    if (user.isStudent()) {
8.        price += 30;   // should be -= 30
9.    }
10.    return price;
11.}
```

L2

L8

测试

✘ t_3

In: Student

Expected: 70

Out: 130

被测程序

测试

✘ t_4

In: Student with Premium and Coupon

Expected: 40

Out: 100

```
1.int calculatePrice(User user) {
2.    int price = 100;
3.    if (user.isPremium())
4.        price -= 10;
5.    if (user.hasCoupon())
6.        price -= 20;
7.    if (user.isStudent()) {
8.        price += 30;   // should be -= 30
9.    }
10.    return price;
11.}
```

L2 L4 L6 L8

被测程序

```
1.int calculatePrice(User user) {
2.    int price = 100;
3.    if (user.isPremium())
4.        price -= 10;
5.    if (user.hasCoupon())
6.        price -= 20;
7.    if (user.isStudent()) {
8.        price += 30;   // should be -= 30
9.    }
10.    return price;
11.}
```

测试

t_1 ✓	t_2 ✓	t_3 ✗	t_4 ✗
In: Normal	In: Premium	In: Student	In: Student with Premium and Coupon
Expected: 100	Expected: 90	Expected: 70	Expected: 40
Out: 100	Out: 90	Out: 130	Out: 100
L2	L2	L2	L2
	L4		L4
			L6
		L8	L8

```
1.int calculatePrice(User user) {
2.     int price = 100;
3.     if (user.isPremium())
4.         price -= 10;
5.     if (user.hasCoupon())
6.         price -= 20;
7.     if (user.isStudent()) {
8.         price += 30;   // should be -= 30
9.    }
10.     return price;
11.}
```

测试	L2	L4	L6	L8
t1	✓			
t2	✓	✓		
t3	✓			✓
t4	✓	✓	✓	✓

可疑度

```
1.int calculatePrice(User user) {
2.     int price = 100;
3.     if (user.isPremium())
4.         price -= 10;
5.     if (user.hasCoupon())
6.         price -= 20;
7.     if (user.isStudent()) {
8.         price += 30;   // should be -= 30
9.    }
10.     return price;
11.}
```

测试	L2	L4	L6	L8
t1	✓			
t2	✓	✓		
t3	✓			✓
t4	✓	✓	✓	✓
可疑度	0.707	0.500	0.707	**1.000**

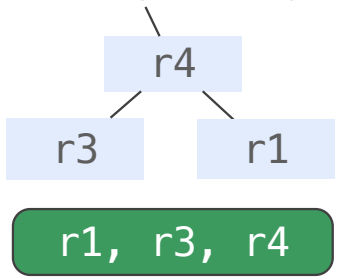
Ochiai(e) = F(e) / √(F·(F(e)+P(e)))

F(e)、P(e)：覆盖 e 的失败 / 通过测试数；F：失败测试总数

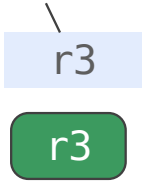
L8: 2 / √(2 × 2) = 1.000

回到 Datalog：事实即测试用例

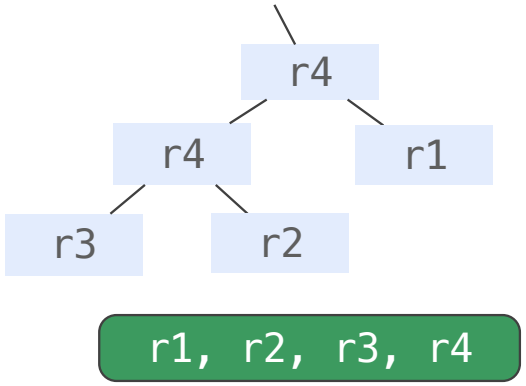
reachable(Main.init)



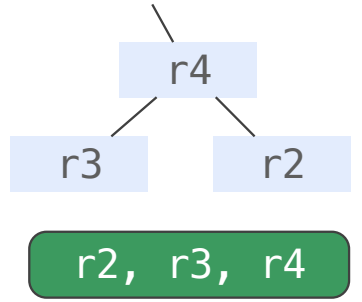
reachable(Main.main)



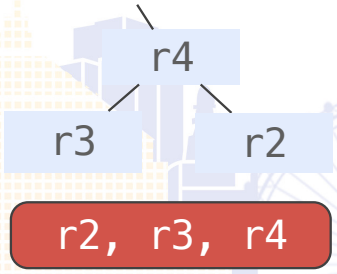
reachable(Main.loadConfig)



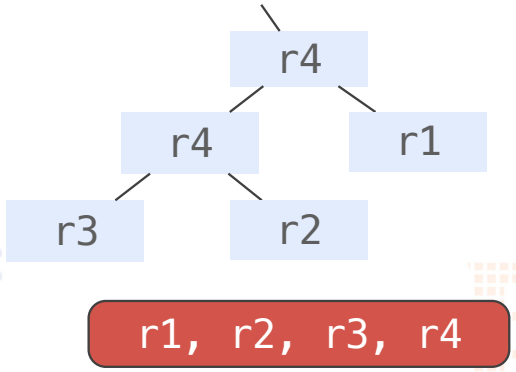
reachable(JsonParser.parse)



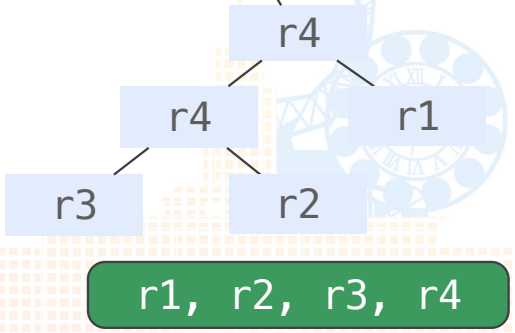
reachable(YamlParser.parse)



reachable(YamlParser.readYaml)



reachable(JsonParser.readJson)



可疑度

输出事实	r1	r2	r3	r4
reachable(Main.main)			✓	
reachable(Main.init)	✓		✓	✓
reachable(Main.loadConfig)	✓		✓	✓
reachable(JsonParser.parse)		✓	✓	✓
reachable(JsonParser.readJson)	✓	✓	✓	✓
reachable(YamlParser.parse)		✓	✓	✓
reachable(YamlParser.readYaml)	✓	✓	✓	✓
可疑度	0.354	**0.707**	0.535	0.577

r2: $2 / \sqrt{(2 \times (2 + 2))} = 0.707$ **最高**

出现在 2 个错误事实和 2 个正确事实的证明树里

SBFL：简史与质疑

简史

1997 Reps, Ball, Das, Larus

为千年虫问题提出程序谱（program spectra），比较不同运行的执行谱

2002 Tarantula（Jones, Harrold, Stasko）

用失败 / 通过测试的覆盖差异给语句打可疑度

2006 Ochiai（Abreu et al.）

更好的可疑度公式；SBFL 成为最常用的缺陷定位技术之一

> **统计调试本质上只是一种启发式方法，不 sound。**
>
> —— 一位 PL 背景的审稿人（大意）

对命令式程序，这个质疑成立；那么对 Datalog 呢?

SBFL 会不会其实特别适合 Datalog?

目前更多是一种直觉

覆盖相同，相关性不同

测试	L8	L9	L10
t1			
t2			
t3	✓	✓	✓
t4	✓	✓	✓

```
1.int calculatePrice(User user) {
2.     int price = 100;
3.     if (user.isPremium())
4.         price -= 10;
5.     if (user.hasCoupon())
6.         price -= 20;
7.     if (user.isStudent()) {
8.         price += 30;   // should be -= 30
9.         logStudentDiscount(user);
10.        recordPromotion(user);
11.    }
12.    return price;
13.}
```

总是伴随失败出现

切片

```
1. int calculatePrice(User user) {
2.     int price = 100;
3.     if (user.isPremium())
4.         price -= 10;
5.     if (user.hasCoupon())
6.         price -= 20;
7.     if (user.isStudent()) {
8.         price += 30;   // should be -= 30
9.         logStudentDiscount(user);
10.        recordPromotion(user);
11.    }
12.    return price;
13.}
```

因果无关

切片

```
1. int calculatePrice(User user) {
2.     int price = 100;
3.     if (user.isPremium())
4.         price -= 10;
5.     if (user.hasCoupon())
6.         price -= 20;
7.     if (user.isStudent()) {
8.         price += 30;   // should be -= 30
9.         logStudentDiscount(user);
10.        recordPromotion(user);
11.    }
12.    return price;
13.}
```

因果无关

- Datalog 天然自带一种切片

方法总览

把推导出的事实当作测试用例，把证明树当作执行谱

有缺陷的 Datalog 程序 + 输入
例如在 Soufflé 上运行的 Doop

→

推导出的事实： 错误 vs. 正确
错误由开发者标出，其余视为正确

→

每个事实的证明树 → 规则谱
每条规则在一个事实中只计一次

→

阶段一：SBFL → 规则排序
Ochiai, Tarantula, Jaccard, Op, InfoGain

→

阶段二：SWPG → 谓词排序
用规则可疑度为谓词依赖加权

命令式程序中的 SBFL	Datalog 中的 SBFL
测试用例	推导出的事实
失败 / 通过的测试	错误 / 正确的事实
程序元素（语句）	规则
测试的覆盖信息	事实的证明树中出现的规则

规则可疑度（以 Ochiai 为例）

$$Ochiai(r) = F(r) / \sqrt{F \times (F(r)+P(r))}$$

F(r)、P(r)：证明树中含有规则 r 的错误 / 正确事实数
F：错误事实总数

CONTENTS | 目录

01 Datalog简介
02 证明树调试
03 新视角：面向 Datalog 的 SBFL
04 基准构建
05 实验评估与总结

基准构建

- 怎么知道它真的有效?
- 需要带 ground truth 的真实 Datalog 缺陷
  - 但 Datalog 调试此前没有标准的基准

大规模的真实 Datalog 代码，开发历史超过十年

```
Fix precise-generics multi-phase analysis
```

提交信息中含有 "Fix ..."

master · doop-completeness ··· 4.20.40 — 1 parent 8bf5781 commit a73b6e8

4 files changed — +88 -87

Filter files...

- souffle-logic
  - analyses/adaptive-2-object-...
    - analysis.dl
  - main
    - context-sensitivity-declara...
    - context-sensitivity.dl
  - src/main/groovy/org/clyze/d...
    - SouffleGenericsMultiPhaseA...

Search within code

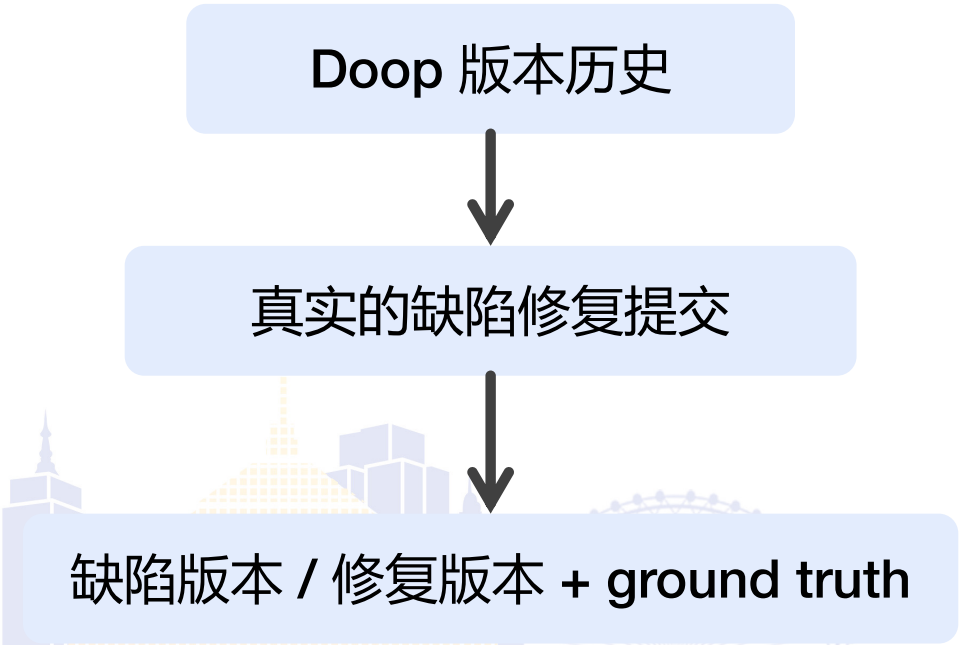
souffle-logic/analyses/adaptive-2-object-sensitive+heap/analysis.dl +2 -1

souffle-logic/main/context-sensitivity-declarations.dl +21 -2

```
@@ -55,8 +55,8 @@
55 55
56 56   .decl OptStoreArrayIndex(?from:Var, ?base:Var)
57 57
58      - //.decl OptArgAssign(?toCtx:configuration.Context, ?to:Var, ?fromCtx:configuration.Context, ?from:Var, ?allowedType:Type)
59      - .decl OptArgAssign(?toCtx:configuration.Context, ?to:Var, ?fromCtx:configuration.Context, ?from:Var)
   58   + .decl OptArgAssign(?toCtx:configuration.Context, ?to:Var, ?fromCtx:configuration.Context, ?from:Var, ?allowedType:Type)
   59   + //.decl OptArgAssign(?toCtx:configuration.Context, ?to:Var, ?fromCtx:configuration.Context, ?from:Var)
60 60   #ifndef INFORMATION_FLOW
61 61   .decl OptReturnAssign(?toCtx:configuration.Context, ?to:Var, ?fromCtx:configuration.Context, ?from:Var)
62 62   #endif // INFORMATION_FLOW
@@ -99,3 +99,22 @@
99 99   // Is the ValueToRefine the real list of objects needing
100 100  // refinement or its negation?
101 101  // .decl NegativeObjectFilter(?v:symbol)
```

基准构建

- 怎么知道它真的有效?
- 需要带 ground truth 的真实 Datalog 缺陷
  - 但 Datalog 调试此前没有标准的基准



- 最终:
  - 37 个不同的缺陷
  - 96 个基准实例

基准构建

- 构建基准不只是比较版本差异

基准构建

- 构建基准不只是比较版本差异

让旧版本 Doop 重新跑起来

- 依赖失效
- 构建 / 兼容性问题
- 网络与仓库问题

基准构建

- 构建基准不只是比较版本差异

让旧版本 Doop 重新跑起来

- 依赖失效
- 构建 / 兼容性问题
- 网络与仓库问题

在语义上对齐执行结果

- 内部表示不同
- 符号表 / ID 不同
- 运行时表示复杂

基准构建

- 构建基准不只是比较版本差异

让旧版本 Doop 重新跑起来

- 依赖失效
- 构建 / 兼容性问题
- 网络与仓库问题

在语义上对齐执行结果

- 内部表示不同
- 符号表 / ID 不同
- 运行时表示复杂

重建缺陷与 ground truth

- 差异 ≠ 缺陷
- 分离出引入缺陷的修改
- 拆分多个相互独立的缺陷

CONTENTS | 目录

01 Datalog简介
02 证明树调试
03 新视角：面向 Datalog 的 SBFL
04 基准构建
05 实验评估与总结

规则定位效果好

来自 Doop 的 96 个真实缺陷实例 · 3 种分析配置 · 5 种 SBFL 公式中最好的 Ochiai

缺陷规则几乎总是排在最前面



3 种配置下的范围

前 10 名只占程序全部规则的 1–2%

哪些缺陷能定位?

缺陷类型	规则级	谓词级
谓词签名错误 头部谓词名、参数个数或顺序写错	≈ 100%	≈ 100%
规则中少写了谓词 最常见的缺陷类型	85–93%	0–7%
常量写错 例如字符串常量写错	80–90%	40–73%

Hit@3（Ochiai 等公式，3 种配置）

规则级是主要信号；谓词级用于精确定位签名类错误

是否实用?

无需标注所有错误事实

10

个已标注的错误事实就够了

错误事实 ≤ 1,000 时，前 3 名命中率 ≈ 90–100%；少量误标影响很小

能在真实程序上运行

8–24 min

真实程序上的端到端耗时

推导事实达上千万；开销近似线性增长

局限与未来工作

局限

- 只导致事实缺失的缺陷：没有推导，就没有谱
- 谓词级：缺失的谓词不会留下痕迹
- 只在 Doop（3 种配置）上做了评估；其他领域的 Datalog 程序有待验证

未来工作

- 结合程序分析，处理遗漏类缺陷
- 改进缺陷谓词定位

总结

1 **证明树太大**

真实缺陷的证明树涉及上万个事实，逐棵手工排查不可行

2 **同时利用多棵证明树**

把推导出的事实当作测试用例，把证明树当作执行谱

3 **SBFL 在命令式程序上受限**

控制流会把无关语句和缺陷绑在一起，覆盖相同不代表相关

4 **SBFL 可能更适合 Datalog**

规则只有真正参与推导才出现在证明树中，信号更干净

谢谢大家！

论文

开源实验材料