



软件分析

# 多角度理解数据流分析

熊英飞

北京大学



# Datalog

- Datalog——逻辑编程语言Prolog的子集
- 一个Datalog程序由如下规则组成：
  - `predicate1(Var or constant list) :- predicate2(Var or constant list), predicate3(Var or constant list), ...`
  - `predicate(constant list)`
- 如：
  - `grandmentor(X, Y) :- mentor(X, Z), mentor(Z, Y)`
  - `mentor(kongzi, mengzi)`
  - `mentor(mengzi, xunzi)`
- Datalog程序的语义
  - 反复应用规则，直到推出所有的结论——即不动点算法
  - 上述例子得到`grandmentor(kongzi, xunzi)`



# 逻辑规则视角

- 一个Datalog编写的正向数据流分析标准型，假设并集
  - $\text{data}(D, V) :- \text{gen}(D, V)$
  - $\text{data}(D, V) :- \text{edge}(V', V), \text{data}(D, V'), \text{not\_kill}(D, V)$
  - $\text{data}(d, \text{entry}) // \text{if } d \in I$
  - $V$ 表示结点， $D$ 表示一个集合中的元素



# 练习：交集的情况怎么写？

- $\text{data}(D, V) :- \text{gen}(D, V)$
- $\text{data}(D, v) :- \text{data}(D, v_1), \text{data}(D, v_2), \dots, \text{data}(D, v_n),$   
 $\text{not\_kill}(D, v) // v_1, v_2, \dots, v_n$ 是 $v$ 的前驱结点
- $\text{data}(d, \text{entry}) // \text{if } d \in I$



# 历史

- 大量的静态分析都可以通过Datalog简洁实现，但因为逻辑语言的效率，一直没有普及
- 2005年，斯坦福Monica Lam团队开发了高效Datalog解释器bddbdb，使得Datalog执行效率接近专门算法的执行效率
- 之后大量静态分析直接采用Datalog实现



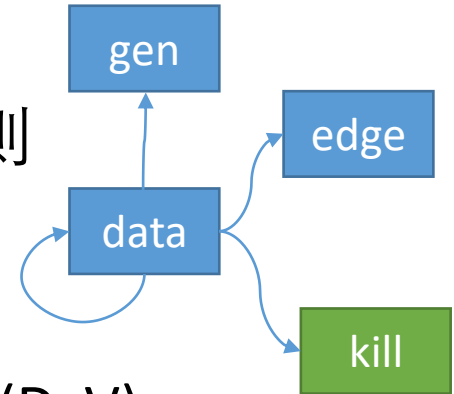
# Datalog $\neg$

- not\_kill关系的构造效率较低
- 理想写法：
  - $\text{data}(D, V) :- \text{edge}(V', V), \text{data}(D, V'), \text{not kill}(D, V)$
- 但是，引入not可能带来矛盾
  - $p(x) :- \text{not } p(x)$
  - 不动点角度理解： 单次迭代并非一个单调函数



# Datalog $\neg$

- 解决方法：分层(stratified)规则
  - 谓词上的任何环状依赖不能包含否定规则
- 依赖示例
  - $\text{data}(D, V) :- \text{gen}(D, V)$
  - $\text{data}(D, V) :- \text{edge}(V', V), \text{data}(D, V'), \text{not kill}(D, V)$
  - $\text{data}(d, \text{entry})$
- 不动点角度理解：否定规则将谓词分成若干层，每层需要计算到不动点，多层之间顺序计算
- 主流Datalog引擎通常支持Datalog  $\neg$





# Datalog引擎

- Souffle
- LogicBlox
- IRIS
- XSB
- Coral
- 更多参考：<https://en.wikipedia.org/wiki/Datalog>





# 方程求解

- 数据流分析的传递函数和 $\sqcap$ 操作定义了一组方程
  - $D_{v_1} = F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - $D_{v_2} = F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
  - ...
  - $D_{v_n} = F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
- 其中
  - $F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n}) = f_{v_1}(I)$
  - $F_{v_i}(D_{v_1}, D_{v_2}, \dots, D_{v_n}) = f_{v_i}(\sqcap_{j \in \text{pred}(i)} D_{v_j})$
- 数据流分析即为求解该方程的最大解
  - 传递函数和 $\sqcap$ 操作表达了该分析的安全性条件，所以该方程的解都是安全的
  - 最大解是最有用的解



# 方程组求解算法

- 在数理逻辑学中，该类算法称为Unification算法
  - 参考：  
[http://en.wikipedia.org/wiki/Unification\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Unification_(computer_science))
- 对于单调函数和有限格，标准的Unification算法就是我们学到的数据流分析算法
  - 从 $(I, T, T, \dots, T)$ 开始反复应用 $F_{v_1}$ 到 $F_{v_n}$ ，直到达到不动点
  - 增量优化：每次只执行受到影响的 $F_{v_i}$

# 术语-流敏感(flow-sensitivity)



- 流非敏感分析 (flow-insensitive analysis) : 如果把程序中语句随意交换位置 (即: 改变控制流), 如果分析结果始终不变, 则该分析为流非敏感分析。
- 流敏感分析 (flow-sensitive analysis) : 其他情况
- 数据流分析通常为流敏感的



# 流非敏感分析

- 转换成同样的方程组，并用不动点算法求解

```
a=100;  
if(a>0)  
  a=a+1;  
b=a+1;
```

流非敏感符号分析

$$a = a \sqcap \text{正} \sqcap a + \text{正}$$
$$b = b \sqcap a + \text{正}$$

不考虑位置，用所有赋值语句更新所有变量

流非敏感活跃变量分析

$$\text{DATA} = \text{DATA} \cup \{a\}$$

对于整个程序产生一个集合，只要程序中有读取变量 $v$ 的语句，就将其加入集合



# 时间空间复杂度

- 活跃变量分析：语句数为 $n$ ，程序中变量个数为 $m$ ，使用bitvector表示集合
- 流非敏感的活跃变量：每条语句对应一个并集操作，时间为 $O(m)$ ，迭代一轮即收敛，因此时间复杂度上界为 $O(nm)$ ，空间复杂度上界为 $O(m)$
- 流敏感的活跃变量分析：格的高度为 $O(m)$ ，即每个结点的值最多变化 $O(m)$ 次。每个结点有最多 $O(n)$ 个后继节点，即每个结点的值最多被更新 $O(mn)$ 次。每次有后继结点变化可以只合并变化的结点，因此单个均摊之后结点总更新复杂度 $O(nm^2)$ ，总时间复杂度上界 $O(n^2m^2)$ ，空间复杂度上界为 $O(nm)$
- 对于特定分析，流非敏感分析能到达很快的处理速度和可接受的精度（如基于SSA的指针分析）



# 程序分析的分类-敏感性

- 一般而言，抽象过程中考虑的信息越多，程序分析的精度就越高，但分析的速度就越慢
- 程序分析中考虑的信息通常用敏感性来表示
  - 流敏感性flow-sensitivity
  - 路径敏感性path-sensitivity
  - 上下文敏感性context-sensitivity
  - 字段敏感性field-sensitivity
- 注意区别：
  - 敏感性 vs 分析结果的形式



# 只返回一组范围的分析

```
If (...)  
    x = 0;  
    y = x;  
else  
    x = 1;  
    y = x - 1;
```

- 求程序执行过程中x和y所有可能取值的范围
- 流敏感分析:  $x:[0, 1]$ ,  $y:[0, 0]$
- 流非敏感分析:  $x:[0, 1]$ ,  $y:[-1, 1]$



# 路径敏感性

- 路径非敏感分析：不考虑程序中的路径可行性，忽略分支循环语句中的条件
- 路径敏感分析：考虑程序中的路径可行性，尽量只分析可能的路径





# 路径敏感vs路径非敏感

```
int f(int x){  
    if (x > 5)  
        x = 10;  
    return x;  
}
```

假设输入的区间为(5, 10]

- 使用路径非敏感的区间分析，得到函数的返回值为
  - (5, 10]
- 使用路径敏感的区间分析，得到函数的返回值为
  - [10, 10]

# 给数据流分析添加基本路径敏感性



```
int f(int x){
  if (x > 5) {
    assert(x > 5);
    x = 10; }
  else {
    assert(x <= 5);
  }
  return x;
}
```

假设输入区间为(5, 10]

- 基本思路:
  - 给每条分支添加assert语句
  - assert语句负责过滤掉必然不可能到达当前分支的抽象状态
- assert(x>5)转换函数
  - $f(x) = x \cap (5, +\infty)$
- assert(x<=5)转换函数
  - $f(x) = x \cap (-\infty, 5]$



# 练习：用学过的Widening & Narrowing 算子进行路径敏感的区域分析

```
x=1;  
while (x < 100) {  
    x++;  
}
```



# Widening & Narrowing 路径 敏感区间分析例子

<code>x=1;</code>	<code>[1,1]</code>
<code>while (x &lt; 100) {</code>	<code>[1, 1]</code>
<code>assert(x &lt; 100);</code>	<code>[1, 1]</code>
<code>x++;</code>	<code>[2, 2]</code>
<code>}</code>	
<code>assert(x &gt;= 100);</code>	<code>∅</code>



# Widening & Narrowing 路径 敏感区间分析例子

<code>x=1;</code>	<code>[1,1]</code>
<code>while (x &lt; 100) {</code>	<code>[1, +∞]</code>
<code>assert(x &lt; 100);</code>	<code>[1, 1]</code>
<code>x++;</code>	<code>[2, 2]</code>
<code>}</code>	
<code>assert(x &gt;= 100);</code>	<code>∅</code>



# Widening & Narrowing 路径 敏感区间分析例子

<code>x=1;</code>	<code>[1,1]</code>
<code>while (x &lt; 100) {</code>	<code>[1, +∞]</code>
<code>assert(x &lt; 100);</code>	<code>[1, 1]</code>
<code>x++;</code>	<code>[2, 2]</code>
<code>}</code>	
<code>assert(x &gt;= 100);</code>	<code>[100, +∞]</code>



# Widening & Narrowing 路径 敏感区间分析例子

<code>x=1;</code>	<code>[1,1]</code>
<code>while (x &lt; 100) {</code>	<code>[1, +∞]</code>
<code>assert(x &lt; 100);</code>	<code>[1, 99]</code>
<code>x++;</code>	<code>[2, 2]</code>
<code>}</code>	
<code>assert(x &gt;= 100);</code>	<code>[100, +∞]</code>



# Widening & Narrowing 路径 敏感区间分析例子

<code>x=1;</code>	<code>[1,1]</code>
<code>while (x &lt; 100) {</code>	<code>[1, +∞]</code>
<code>assert(x &lt; 100);</code>	<code>[1, 99]</code>
<code>x++;</code>	<code>[2, 100]</code>
<code>}</code>	
<code>assert(x &gt;= 100);</code>	<code>[100, +∞]</code>





# Widening & Narrowing 路径 敏感区间分析例子

<code>x=1;</code>	<code>[1,1]</code>
<code>while (x &lt; 100) {</code>	<code>[1, 100]</code>
<code>assert(x &lt; 100);</code>	<code>[1, 99]</code>
<code>x++;</code>	<code>[2, 100]</code>
<code>}</code>	
<code>assert(x &gt;= 100);</code>	<code>[100,100]</code>



# 路径敏感的数据流分析

- 优点：
  - 完全兼容已有数据流分析框架
  - 利用已有技术直接支持循环、过程间等复杂分析
- 缺点：
  - 很多条件类型无法写出过滤函数
    - 给定条件 $x > 5$ ，如何做reaching definition分析？
    - 给定条件 $x > y$ ，如何做路径敏感的区间分析？
  - 同标准数据流分析类似，很多情况无法精确判断
    - `if (y > 0) x = 10; else x=1; if (2<x && x<6) x=20;`
    - $x$ 不会赋值成20，但是区间分析判断不出来
  - 无法分路径输出结果、无法回答“什么输入会导致除零错误”的问题
- 后续课程将介绍符号执行技术，和数据流分析一定程度互补



# 作业:

- 下载任意Datalog引擎，用Datalog编写下面程序的符号分析，提交程序和运行截图
  - 注：只是将如下程序手动转换成Datalog规则，不用编写针对任意程序通用的分析工具

```
x*=-100;  
y+=1;  
while(y < z) {  
    x *= -100;  
    y += 1;  
}
```

输入：x为负，y为零，z为正  
求输出的符号



# 参考资料

- Lecture Notes on Static Analysis
  - <https://cs.au.dk/~amoeller/spa/>
- Datalog Introduction
  - Jan Chomicki
  - <https://cse.buffalo.edu/~chomicki/636/datalog-h.pdf>
- Datalog引擎列表
  - <https://en.wikipedia.org/wiki/Datalog>