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
  - data(D, V) :- gen(D, V)
  - data(D, V) :- edge(V’, V), data(D, V’), not_kill(D, V)
  - data(d, entry) // if d ∈ 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?

- data(D, V) :- gen(D, V)
- data(D, v) :- data(D, v_1), data(D, v_2), …, data(D, v_n), not_kill(D, v) //$v_1, v_2, \ldots v_n$是v的前驱结点
- data(d, entry) // if $d \in I$

历史

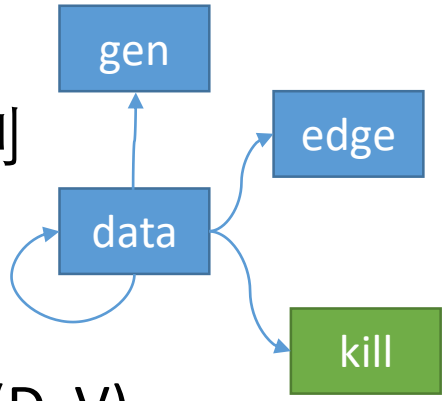
- 大量的静态分析都可以通过Datalog简洁实现，但因为逻辑语言的效率，一直没有普及
- 2005年，斯坦福Monica Lam团队开发了高效Datalog解释器bddbddb，使得Datalog执行效率接近专门算法的执行效率
- 之后大量静态分析直接采用Datalog实现

Datalog¬

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

Datalog¬

- 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
  - data(D, V) :- gen(D, V)
  - data(D, V) :- edge(V’, V), data(D, V’), not kill(D, V)
  - data(d, 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¬

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⊓操作定义了一组方程
  - $\mathrm{D}_{\mathrm{v}_1} = \mathrm{F}_{\mathrm{v}_1}(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \ldots, \mathrm{D}_{\mathrm{v}_n})$
  - $\mathrm{D}_{\mathrm{v}_2} = \mathrm{F}_{\mathrm{v}_2}(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \ldots, \mathrm{D}_{\mathrm{v}_n})$
  - …
  - $\mathrm{D}_{\mathrm{v}_n} = \mathrm{F}_{\mathrm{v}_n}(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \ldots, \mathrm{D}_{\mathrm{v}_n})$
- 其中
  - $F_{v_1}(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \ldots, \mathrm{D}_{\mathrm{v}_n}) = f_{v_1}(I)$
  - $F_{v_i}(\mathrm{D}_{\mathrm{v}_1}, \mathrm{D}_{\mathrm{v}_2}, \ldots, \mathrm{D}_{\mathrm{v}_n}) = f_{v_i}\left(\sqcap_{j \in pred(i)} D_{v_j}\right)$
- 数据流分析即为求解该方程的最大解
  - 传递函数和⊓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)
- 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$(I, \top, \top, \ldots, \top)$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

$a = a \sqcap 正 \sqcap a + 正$

$b = b \sqcap a + 正$

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

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

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)，转移函数、交汇运算和比较运算都是O(m)，均摊分析更新单个结点需要O(1)次上述操作，时间复杂度上界为O(nm^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，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;
}
```

假设输入的区间为(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间分析


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

Widening & Narrowing路径敏感区间分析例子

x=1;	[1,1]
while (x < 100) {	[1, 1]
assert(x < 100);	[1, 1]
x++;	[2, 2]
}	
assert(x >= 100);	∅

Widening & Narrowing路径敏感区间分析例子

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

Widening & Narrowing路径敏感区间分析例子

x=1;	[1,1]
while (x < 100) {	[1, +∞]
assert(x < 100);	[1, 1]
x++;	[2, 2]
}	
assert(x >= 100);	$[100, +\infty]$

Widening & Narrowing路径敏感区间分析例子

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

Widening & Narrowing路径敏感区间分析例子

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

Widening & Narrowing路径敏感区间分析例子


```
x=1;                          [1,1]
while (x < 100) {             [1, 100]
  assert(x < 100);            [1, 99]
  x++;                        [2, 100]
}
assert(x >= 100);             [100,100]
```

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

敏感性 vs 结果精度

- 敏感性主要是指分析过程中是否考虑了某种信息
- 但结果仍然可以在较粗的粒度汇报

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