软件分析

多角度理解数据流分析

熊英飞
北京大学
2017

方程求解

- 数据流分析的传递函数和⊓操作定义了一组方程
  - $\mathrm{D_{v_1}} = \mathrm{F_{v_1}}(\mathrm{D_{v_1}}, \mathrm{D_{v_2}}, \ldots, \mathrm{D_{v_n}})$
  - $\mathrm{D_{v_2}} = \mathrm{F_{v_2}}(\mathrm{D_{v_1}}, \mathrm{D_{v_2}}, \ldots, \mathrm{D_{v_n}})$
  - …
  - $\mathrm{D_{v_n}} = \mathrm{F_{v_n}}(\mathrm{D_{v_1}}, \mathrm{D_{v_2}}, \ldots, \mathrm{D_{v_n}})$
- 其中
  - $F_{v_1}(\mathrm{D_{v_1}}, \mathrm{D_{v_2}}, \ldots, \mathrm{D_{v_n}}) = f_{v_1}(I)$
  - $F_{v_i}(\mathrm{D_{v_1}}, \mathrm{D_{v_2}}, \ldots, \mathrm{D_{v_n}}) = f_{v_i}\left(\sqcap_{j \in pred(i)} D_{v_j}\right)$
- 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：每条语句的操作时间为O(m)，因此时间复杂度上界为O(m^2)，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)，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）

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

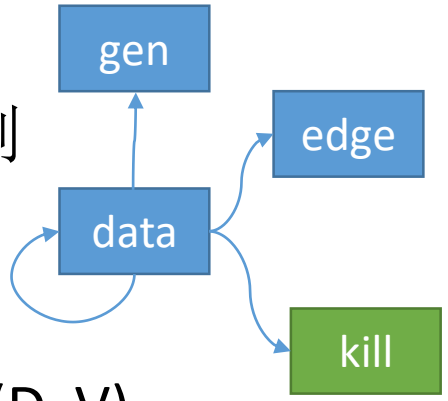
- 大量的静态分析都可以通过Datalog简洁实现，但因为逻辑语言的效率，一直没有普及
- 2005年，斯坦福Monica Lam团队开发了高效Datalog解释器bddbddb，使得Datalog执行效率接近专门算法的执行效率
- 之后大量静态分析直接采用Datalog实现

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

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

```
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