



软件分析

抽象解释和其他模型

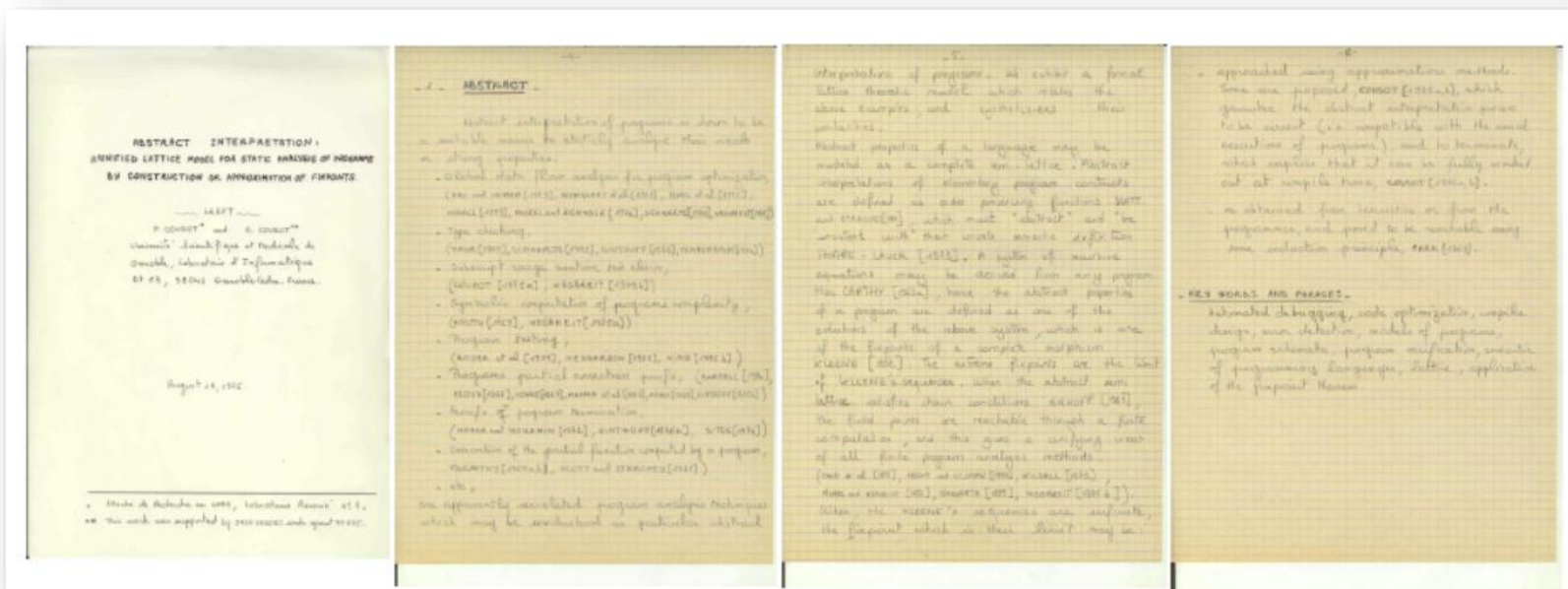
熊英飞

北京大学



抽象解释

- 最早发表于POPL'77（手写的100页论文）





所获荣誉

2013年ACM SIGPLAN 程序语言成就奖

**SIGPLAN**
To explore programming language concepts and tools focusing on design, implementation and efficient use.

[Report Problem](#) [Contact Us](#)

[Home](#) [Awards](#) [Conferences](#) [Resources](#) [Membership](#) [SIGPLAN Research Highlights](#) [Student Information](#) [Publications](#) [Announcements](#)

Programming Languages Achievement Award

Given by ACM SIGPLAN to recognize an individual or individuals who has made a significant and lasting contribution to the field of programming languages. The contribution can be a single event or a life-time of achievement. The award includes a prize of \$5,000. The award is presented at SIGPLAN's [PLDI conference](#) the following June.

Nominations

- [Details of the nomination and award process \(pdf\)](#).
- Please use <http://awards.sigplan.org/> to submit nominations.

Recipients of the Achievement Award

2013: Patrick and Radhia Cousot

Patrick and Radhia Cousot are the co-inventors of abstract interpretation, a unifying theory of sound abstraction and approximation of structures involved in various domains of computer science, such as formal semantics, specification, proof, and verification. In particular, abstract interpretation has had a major impact on the development of the static analysis of software. In their original work, the Cousots showed how to relate a static analysis to a language's standard semantics by means of a second, abstract semantics that makes precise which features of the full language are being modeled and which are being discarded (or abstracted), providing for the first time both a formal definition of and clear methodology for designing and proving the correctness of static analyses. Subsequently, the Cousots contributed many of the building blocks of abstract interpretation in use today, including chaotic iteration, widening, narrowing, combinations of abstractions, and a number of widely used abstract domains. This work has developed a remarkable set of intellectual tools and has found its way into practice in the form of widely used libraries and frameworks. Finally, the Cousots and their collaborators have contributed to demonstrating the utility of static analysis to society. They led the development of the AstrÉE static analyzer, which is used in the medical, automotive, and aerospace industry for verifying the absence of a large class of common programming errors in low-level embedded systems code. This achievement stands as one of the most substantial successes of program verification to date.



所获荣誉

2018年 约翰·冯·诺依曼奖章



Patrick Cousot awarded John von Neumann Medal

Patrick Cousot is the recipient of the IEEE John von Neumann medal, given "for outstanding achievements in computer-related science and technology".

[Read More](#)



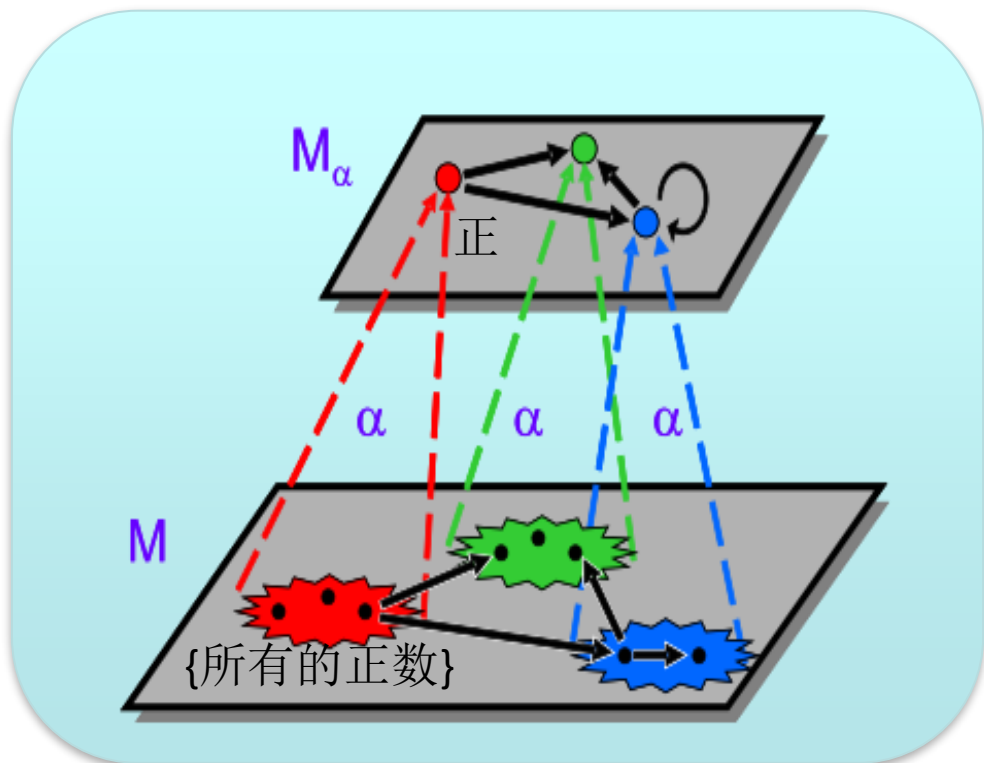
IEEE JOHN VON NEUMANN MEDAL RECIPIENTS

2018 PATRICK COUSOT
Professor, New York University,
New York, New York, USA

"For introducing abstract interpretation, a powerful framework for automatically calculating program properties with broad application to verification and optimization."

抽象解释

- 主要解释抽象空间和具体空间的关系



抽象空间

具体空间



抽象解释

- 具体化函数 γ 将抽象值映射为具体域的元素
 - 本课程中，具体域的元素为具体值的集合
 - $\gamma(\text{正}) = \{\text{所有的正数}\}$
 - $\gamma(\perp) = \emptyset$
- 抽象化函数 α 将具体值的集合映射为抽象值
 - $\alpha(\{\text{所有的正数}\}) = \text{正}$
 - $\alpha(\{1, 2\}) = \text{正}$
 - $\alpha(\{-1, 0\}) = \text{赧}$
- 假设抽象域上存在偏序关系 $\sqsubseteq$



伽罗瓦连接

Galois Connection

- 我们称 γ 和 α 构成抽象域**虚**和具体域 **D** 之间的一个伽罗瓦连接，记为

$$(D, \sqsubseteq) \sqsubseteq_{\alpha}^{\gamma} (\text{虚}, \sqsubseteq)$$

- 当且仅当

$$\forall X \in D, \text{甲} \in \text{虚}: \alpha(X) \sqsubseteq \text{甲} \Leftrightarrow X \sqsubseteq \gamma(\text{甲})$$



定理

- $(D, \subseteq) \sqsubseteq_{\alpha}^{\gamma} (\text{虚}, \sqsubseteq)$ 当且仅当以下所有公式成立
- α 是单调的: $\forall X, Y \in D: X \subseteq Y \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$
- γ 是单调的: $\forall \text{甲}, \text{乙} \in \text{虚}: \text{甲} \sqsubseteq \text{乙} \Rightarrow \gamma(\text{甲}) \subseteq \gamma(\text{乙})$
- $\gamma \circ \alpha$ 保持或增大输入: $\forall X \in D: X \subseteq \gamma(\alpha(X))$
- $\alpha \circ \gamma$ 保持或缩小输入: $\forall \text{甲} \in \text{虚}: \alpha(\gamma(\text{甲})) \sqsubseteq \text{甲}$

程序静态分析中，一般直接将 $\alpha \circ \gamma$ 定义为等价变换，数学上称为“伽罗瓦插入”。

15 非伽罗瓦插入的情况通常因为抽象域有冗余。比如允许区间 $[3, 2]$ （即空区间）



证明

- $\Rightarrow$
 - $\forall X \in \mathbf{D}: X \subseteq \gamma(\alpha(X))$
 - 由 $\alpha(X) \sqsubseteq \alpha(X)$ 和伽罗瓦连接定义可得 $X \subseteq \gamma(\alpha(X))$
 - $\forall \text{甲} \in \mathbf{虚}: \alpha(\gamma(\text{甲})) \sqsubseteq \text{甲}$
 - 由 $\gamma(\text{甲}) \subseteq \gamma(\text{甲})$ 和伽罗瓦连接定义可得 $\alpha(\gamma(\text{甲})) \sqsubseteq \text{甲}$
 - $\forall X, Y \in \mathbf{D}: X \subseteq Y \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$
 - $X \subseteq Y \subseteq \gamma(\alpha(Y)) \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$
 - $\forall \text{甲}, \text{乙} \in \mathbf{虚}: \text{甲} \sqsubseteq \text{乙} \Rightarrow \gamma(\text{甲}) \subseteq \gamma(\text{乙})$
 - $\alpha(\gamma(\text{甲})) \sqsubseteq \text{甲} \sqsubseteq \text{乙} \Rightarrow \gamma(\text{甲}) \subseteq \gamma(\text{乙})$



证明

• $\Leftarrow$

- $\alpha(X) \sqsubseteq \text{甲}$
- $\Rightarrow \gamma(\alpha(X)) \subseteq \gamma(\text{甲})$ **【 γ 的单调性】**
- $\Rightarrow X \subseteq \gamma(\text{甲})$ **【 $X \subseteq \gamma(\alpha(X))$ 】**

- $X \subseteq \gamma(\text{甲})$
- $\Rightarrow \alpha(X) \sqsubseteq \alpha(\gamma(\text{甲}))$ **【 α 的单调性】**
- $\Rightarrow \alpha(X) \sqsubseteq \text{甲}$ **【 $\alpha(\gamma(\text{甲})) \sqsubseteq \text{甲}$ 】**



函数抽象

- 给定伽罗瓦连接 $(D, \sqsubseteq) \sqsubseteq_{\alpha}^{\gamma} (\text{虚}, \sqsubseteq)$
- 给定 D 上的函数 f 和 虚 上的函数 φ
- φ 是 f 的安全抽象, 当且仅当
 - $(\alpha \circ f \circ \gamma)(\text{甲}) \sqsubseteq \varphi(\text{甲})$
 - 即 $(f \circ \gamma)(\text{甲}) \sqsubseteq (\gamma \circ \varphi)(\text{甲})$
- φ 是 f 的最佳抽象, 当且仅当
 - $\alpha \circ f \circ \gamma = \varphi$
- φ 是 f 的精确抽象, 当且仅当
 - $f \circ \gamma = \gamma \circ \varphi$
- 最佳抽象总是存在, 但精确抽象不一定存在
 - 能举一个精确抽象不存在的例子吗?



定义程序分析的安全性

- 执行踪迹（具体执行序列）：(语句编号,内存状态)构成的序列
- 程序分析：分析程序所有执行踪迹集合的属性
 - 符号分析：所有到达某程序节点的状态中变量的符号
 - 可达定值：执行踪迹集合的所有踪迹在某个节点的可达定值的并
 - 可用表达式：执行踪迹集合中所有踪迹在某个节点的可用表达式的交
 - 定义通常包括两部分：1. 单条踪迹的属性 2. 如何从单条的属性得到集合的属性
- 程序分析的安全性：在分析结果域存在某种偏序关系，与理想结果相同或者更大视为安全



定义程序分析的安全性

- 程序的执行踪迹集合和分析结果域构成伽罗瓦连接

- 具体域：执行踪迹集合+集合子集关系
- **抽象域**：分析结果+分析结果上的偏序关系
- β ：单条轨迹对应的分析结果
- α ：踪迹集合对应的精确分析结果，定义为
 - $\alpha(\emptyset) = \perp$
 - $\alpha(X) = \bigsqcup_{x \in X} \beta(x)$
- $\gamma(\text{甲}) = \{x \mid \beta(x) \sqsubseteq \text{甲}\}$

红色为特定分析需要定义的部分

- 抽象域有限时，容易证明上述元素形成伽罗瓦连接

- $\alpha(X) \sqsubseteq \text{甲} \Leftrightarrow \forall x \in X, \beta(x) \sqsubseteq \text{甲} \Leftrightarrow X \subseteq \gamma(\text{甲})$
- 第一步需要用到下页的最小上界定理
 - 给定集合 $\{\beta(x) \mid x \in X\}$
 - 甲是该集合的上界，而 $\alpha(X)$ 是该集合的最小上界
- 第二步由 γ 定义直接可得



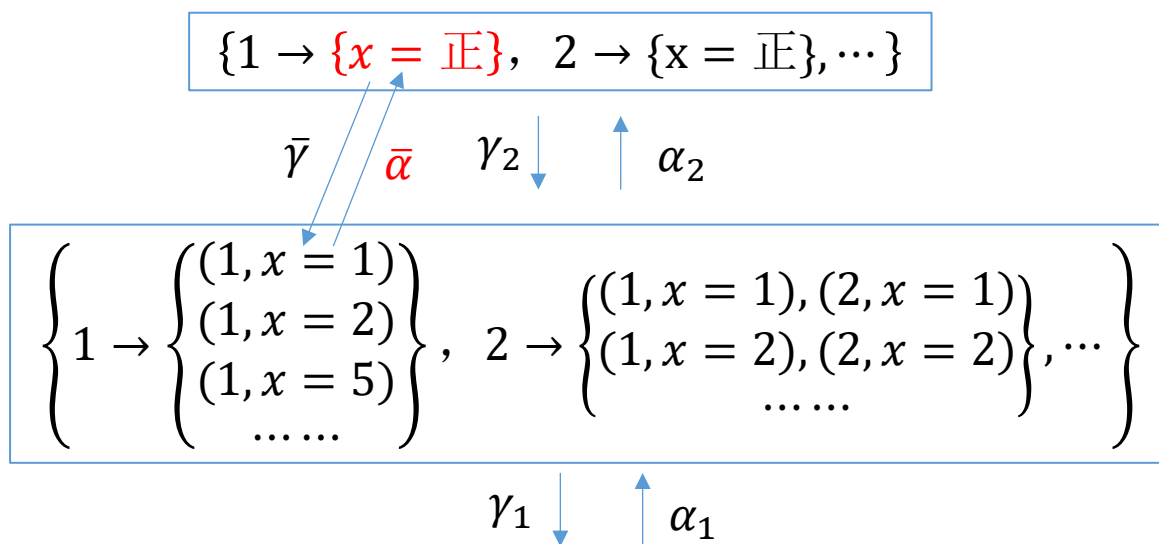
集合的最小上界

- 上界：给定集合 S ，如果满足 $\forall s \in S: s \sqsubseteq u$ ，则称 u 是 S 的一个上界
- 最小上界：设 u 是集合 S 的上界，给定任意上界 u' ，如果满足 $u \sqsubseteq u'$ ，则称 u 是 S 的最小上界
- 引理： $\sqcup_{s \in S} s$ 是有限非空集合 S 的最小上界
 - 证明：
 - 根据幂等性、交换性和结合性，我们有 $\forall v \in S: (\sqcup_{s \in S} s) \sqcup v = \sqcup_{s \in S} s$ ，所以 $\sqcup_{s \in S} s$ 是 S 的上界
 - 给定另一个上界 u ，我们有 $\forall s \in S: s \sqcup u = u$ ， $(\sqcup_{s \in S} s) \sqcup u = (\sqcup_{s \in S} (s \sqcup u)) = u$ ，所以 $\sqcup_{s \in S} s$ 是最小上界



定义数据流分析的安全性

- 数据流分析在每个程序点产生一个结果，可以看做是下面多个伽罗瓦连接的复合

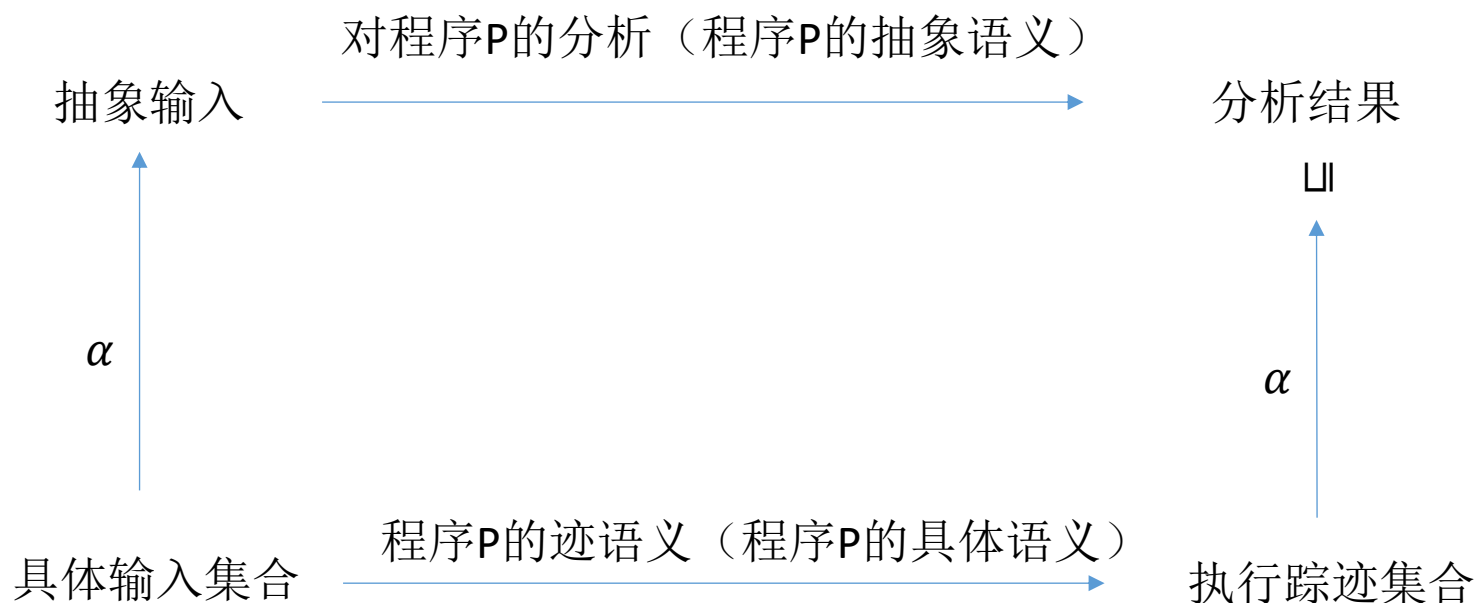


红色为需要定义的部分



迹语义 (Trace Semantics)

- 迹语义是一个函数，将程序和输入集合映射为执行踪迹的集合



也就是说，程序分析是对具体语义的一个安全函数抽象



控制流图的具体语义

- 假设每个控制流节点 v 对应两个函数
 - 具体（内存状态）转换函数：
 - $trans_v: M \rightarrow 2^M$
 - 控制转移函数：
 - $next_v: M \rightarrow 2^{succ(v)}$
 - 其中 M 为所有内存状态的集合
- 以上函数的返回值都是集合，对应不确定的执行，比如返回随机数
- 考虑反向分析的话，需要反向执行程序，程序的反向执行通常也是不确定的



控制流图的具体语义

- 定义如下单步执行函数
 - $Step(T) = \{t + (v', m') \mid$
 $t \in T$
 $v' \in next_v(m),$
 $m' \in trans_{v'}(m),$
 $(v, m) = last(t)\} \cup T$
 - 即每次把序列加长一步
- 那么一个程序的迹语义就是 $Step^\infty$, 定义为
 - $Step^\infty(T) = \bigcup_{k \geq 0} Step^k(T_0)$
 - 其中, $T_0 = \{(Entry, input)\}$, $input$ 是程序的起始状态



控制流图的抽象语义

- 复习：之前定义过轮询函数
 - $F(\text{OUT}_{v_1}, \text{OUT}_{v_2}, \dots, \text{OUT}_{v_n}) =$
 $(f_{v_1}(\sqcup_{w \in \text{pred}(v_1)} \text{OUT}_w),$
 $f_{v_2}(\sqcup_{w \in \text{pred}(v_2)} \text{OUT}_w),$
 $\dots,$
 $f_{v_n}(\sqcup_{w \in \text{pred}(v_n)} \text{OUT}_w))$
- 数据流分析的结果为 $F^\infty(I)$
- 我们现在要证明轮询函数 F 是 Step 的安全抽象



标准数据流分析的安全性

- 如果任意节点 v 上的抽象域转换函数 f_v 满足如下条件
 - $t \in \text{Step}(\bar{\gamma}(\text{甲})) \wedge \text{last}(t).node = v$
 $\Rightarrow t \in \bar{\gamma}(f_v(\text{甲}))$
- 那么 F 是Step的安全函数抽象，即
 - $\text{Step}(\gamma(\text{甲})) \subseteq \gamma(F(\text{甲}))$
- 证明：和之前证明类似，考虑顺序、分支、合并等多种不同情况，路径都仍然在转换之后的抽象值中。略



小结： 基于抽象解释设计程序分析

- 程序的具体语义：
 - 反复应用某种具体单步执行函数得到程序的踪迹集合
- 程序的抽象语义：
 - 针对问题设计抽象域
 - 设计抽象单步执行函数，是具体执行函数的安全抽象
 - 证明抽象单步执行函数收敛
 - 通常基于单调性+半格高度有限
 - 半格无限高度时可以利用加宽



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
 - 抽象域的值可以进一步映射为想要的分析结果

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), 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



流非敏感区间分析举例

```
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: $x:[0, 1]$, $y:[-1, 1]$

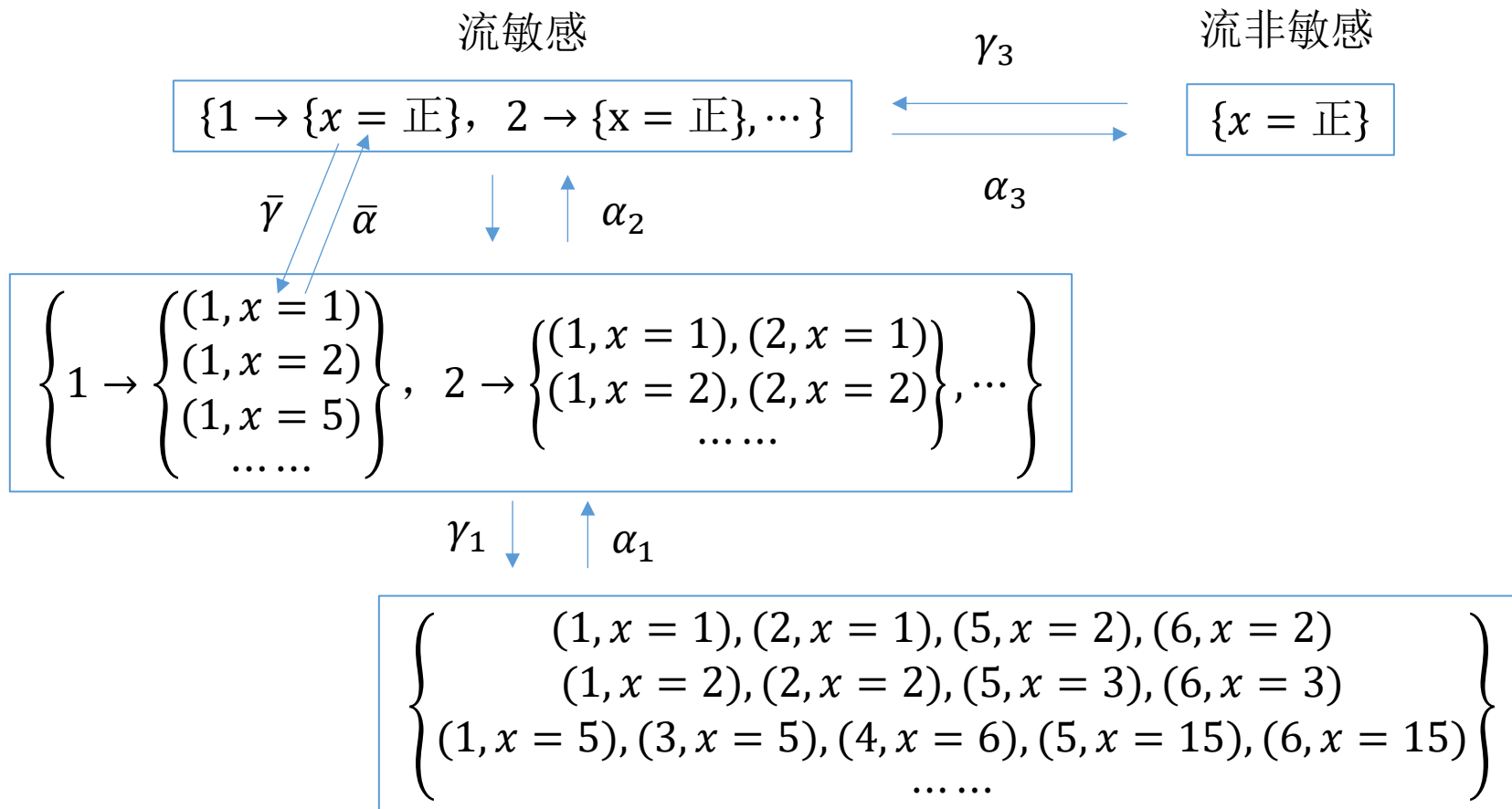


流非敏感分析

- 不区分不同节点上的OUT值，我们就得到了流非敏感分析
 - $F_{fi}(OUT) = \sqcup_{v \in V} f_v(OUT)$
 - 初始OUT值是 $\perp$
- 对比流敏感分析
 - $F(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n}) =$
 $(f_{v_1}(\sqcup_{w \in \text{pred}(v_1)} OUT_w),$
 $f_{v_2}(\sqcup_{w \in \text{pred}(v_2)} OUT_w),$
 $\dots,$
 $f_{v_n}(\sqcup_{w \in \text{pred}(v_n)} OUT_w))$
- 可以定义流非敏感结果和流敏感结果之间的伽罗瓦连接。
 - $\alpha(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n}) = \sqcup_{v \in V} OUT_v$
 - $\gamma(OUT) = (OUT, OUT, \dots, OUT)$
- 容易看出， F_{fi} 是F的安全抽象。



流敏感vs流非敏感





流非敏感分析

- 实际中的流非敏感分析通常针对分析进行适当化简

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

流非敏感符号分析

$$F(a, b) \\ = (a \sqcup \text{正} \sqcup a + \text{正}, \\ b \sqcup a + \text{正})$$

按变量组织转换函数

流非敏感活跃变量分析

$$OUT = OUT \cup \{a\}$$

如果某节点的KILL中的变量在任意节点的GEN中，则该变量永远不会被删除，如果不在任意节点的GEN中，则该变量永远不会被添加。所以可以直接忽略KILL。



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分析：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 $O(m)$ ，即每个节点的值最多变化 $O(m)$ 次。每个节点有最多 $O(n)$ 个后继节点，即每个节点的值最多被更新 $O(mn)$ 次。每次有后继节点变化可以只合并变化的节点，因此单个均摊之后节点总更新复杂度 $O(nm^2)$ ，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，流非敏感分析能达到很快的处理速度和可接受的精度（如基于SSA的指针分析）



路径敏感性

- 路径敏感分析：区分不同路径所能达到的状态
- 路径非敏感分析：忽略不同路径的差别
- 带条件压缩函数的分析在部分文献中会被称为路径敏感分析，但实际上在路径汇合时仍然会丢失路径信息



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：
 - `lineage(X, Y) :- mentor(X, Y)`
 - `lineage(X, Y) :- lineage(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`lineage(kongzi, xunzi)`



从逻辑编程角度看程序分析

- 一个Datalog编写的正向数据流分析标准型，假设并集
 - $\text{out}(D, V) \text{ :- gen}(D, V)$
 - $\text{out}(D, V) \text{ :- edge}(V', V), \text{out}(D, V'), \text{not_kill}(D, V)$
 - $\text{out}(d, \text{entry}) \text{ // if } d \in I$
 - V 表示entry以外的节点， D 表示一个集合中的元素



练习：交集的情况怎么写？

- $\text{out}(D, V) \text{ :- gen}(D, V)$
- $\text{out}(D, v) \text{ :- out}(D, v_1), \text{out}(D, v_2), \dots, \text{out}(D, v_n),$
 $\text{not_kill}(D, v) \text{ // } v_1, v_2, \dots v_n \text{ 是 } v \text{ 的前驱节点}$
- $\text{out}(d, \text{entry}) \text{ // 如果 } d \in I$
- $\text{out}(d, v) \text{ // 对任意值和非entry节点 } v$



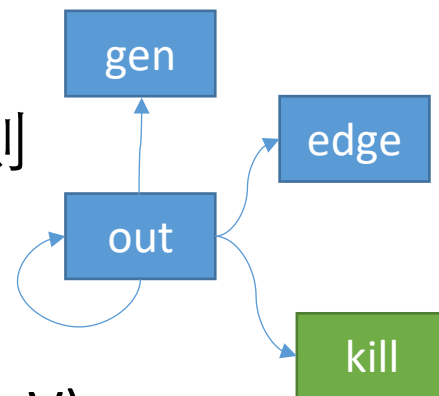
Datalog $\neg$

- not_kill关系的构造效率较低
- 理想写法：
 - $\text{out}(D, V) \text{ :- edge}(V', V), \text{out}(D, V'), \text{not kill}(D, V)$
- 但是，引入not可能带来矛盾
 - $p(x) \text{ :- not } p(x)$
 - 推导式的含义不清楚，没有p属性的元素加上了p属性，就和前提矛盾了



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
 - $\text{out}(D, V) \text{ :- } \text{gen}(D, V)$
 - $\text{out}(D, V) \text{ :- } \text{edge}(V', V), \text{out}(D, V'), \text{not kill}(D, V)$
 - $\text{out}(d, \text{entry})$



- 计算方法：将谓词分成若干层，确保负依赖只能指向低层，正依赖可以指向同层或低层。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>



历史

- 大量的静态分析都可以通过Datalog简洁表示，但逻辑语言的效率一直是瓶颈之一。
- 2005年，斯坦福Monica Lam团队开发了高效Datalog解释器bddbldb，显著提升了Datalog执行效率。
 - 后续工具采用关系索引、增量求值和代码生成等方法继续提高效率。
- 之后大量静态分析直接采用Datalog实现



方程求解

- 数据流分析的传递函数和 $\sqcup$ 操作定义了一组方程
 - $OUT_{v_1} = F_{v_1}(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n})$
 - $OUT_{v_2} = F_{v_2}(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n})$
 - ...
 - $OUT_{v_n} = F_{v_n}(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n})$
- 其中
 - $F_{entry}(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n}) = I$
 - $F_{v_i}(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n}) = f_{v_i}(\sqcup_{w \in pred(v_i)} OUT_w)$
- 数据流分析即为求解该方程的最小解
 - 传递函数和 $\sqcup$ 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
 - 最小解是最精确的解



从不等式到方程组

- 不等式
 - $D_{v_1} \supseteq F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
 - $D_{v_2} \supseteq F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
 - ...
 - $D_{v_n} \supseteq F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
- 可以通过转换成如下方程组求解
 - $D_{v_1} = D_{v_1} \sqcup F_{v_1}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
 - $D_{v_2} = D_{v_2} \sqcup F_{v_2}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$
 - ...
 - $D_{v_n} = D_{v_n} \sqcup F_{v_n}(D_{v_1}, D_{v_2}, \dots, D_{v_n})$



作业：

- 一个用Datalog编写的符号分析，在应用到下面程序上时，产生了一部分规则，请补全剩余的规则，并分析相比之前的符号分析，精度是否一样？如果不一样，请举一个例子。分析规则和结果中不出现 $\bot$ 和 $\perp$ 。
 - 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

```
1. x-=1;  
2. y+=1;  
3. while(y < z) {  
4.  x *= -100;  
5.  y += 1;}
```

输入：x为负，y为零，z为正
求输出的符号

```
out(x, entry, 负)  
out(y, entry, 零)  
out(z, entry, 正)  
out(x, exit, ?)  
out(y, exit, ?)  
out(z, exit, ?)  
edge(1, 2), edge(2, 3),  
edge(5, 3), edge(3, 4),
```

```
edge(4, 5), edge(entry, 1),  
edge(3, exit)  
out(x, 1, 正) :- in(x, 1, 正)  
out(x, 1, 零) :- in(x, 1, 正)  
out(x, 1, 负) :- in(x, 1, 负)  
out(x, 1, 负) :- in(x, 1, 零)  
out(v, 3, 甲) :- in(v, 3, 甲)
```



参考资料

- Datalog Introduction
 - Jan Chomicki
 - <https://cse.buffalo.edu/~chomicki/636/datalog-h.pdf>
- Datalog引擎列表
 - <https://en.wikipedia.org/wiki/Datalog>