



软件分析

抽象解释和分析精度

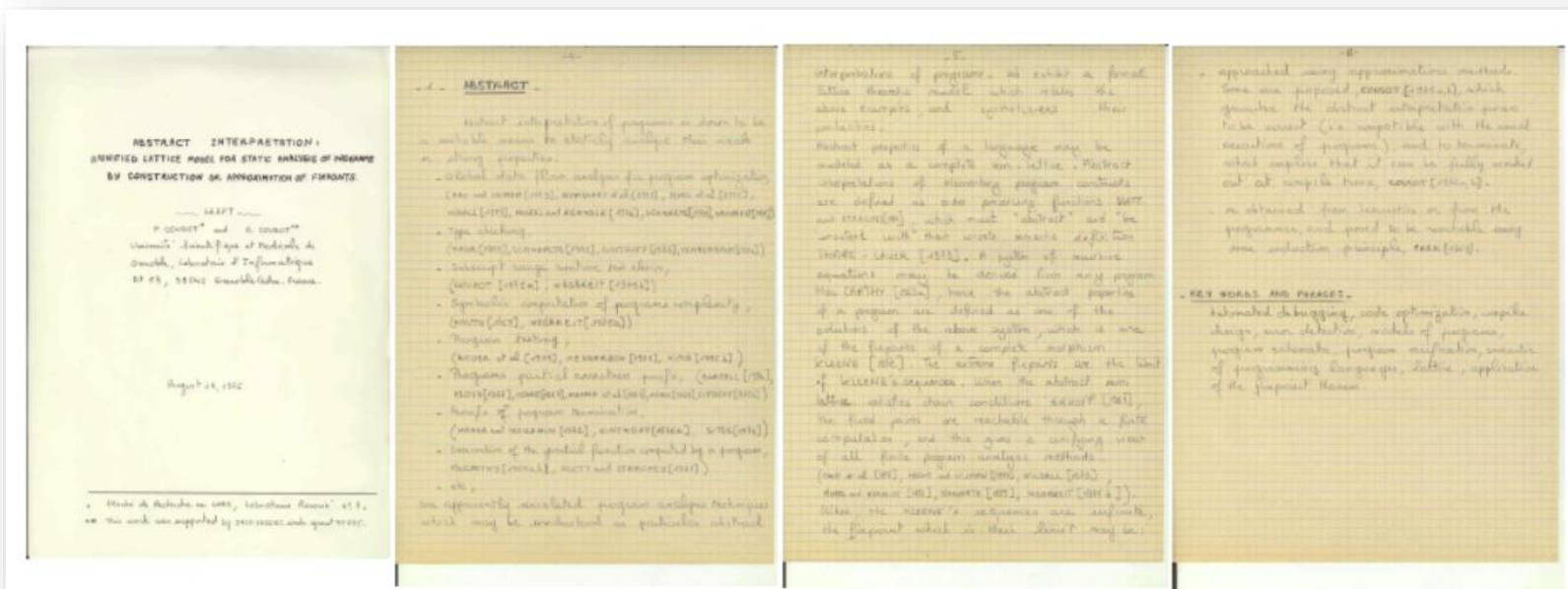
熊英飞

北京大学



抽象解释

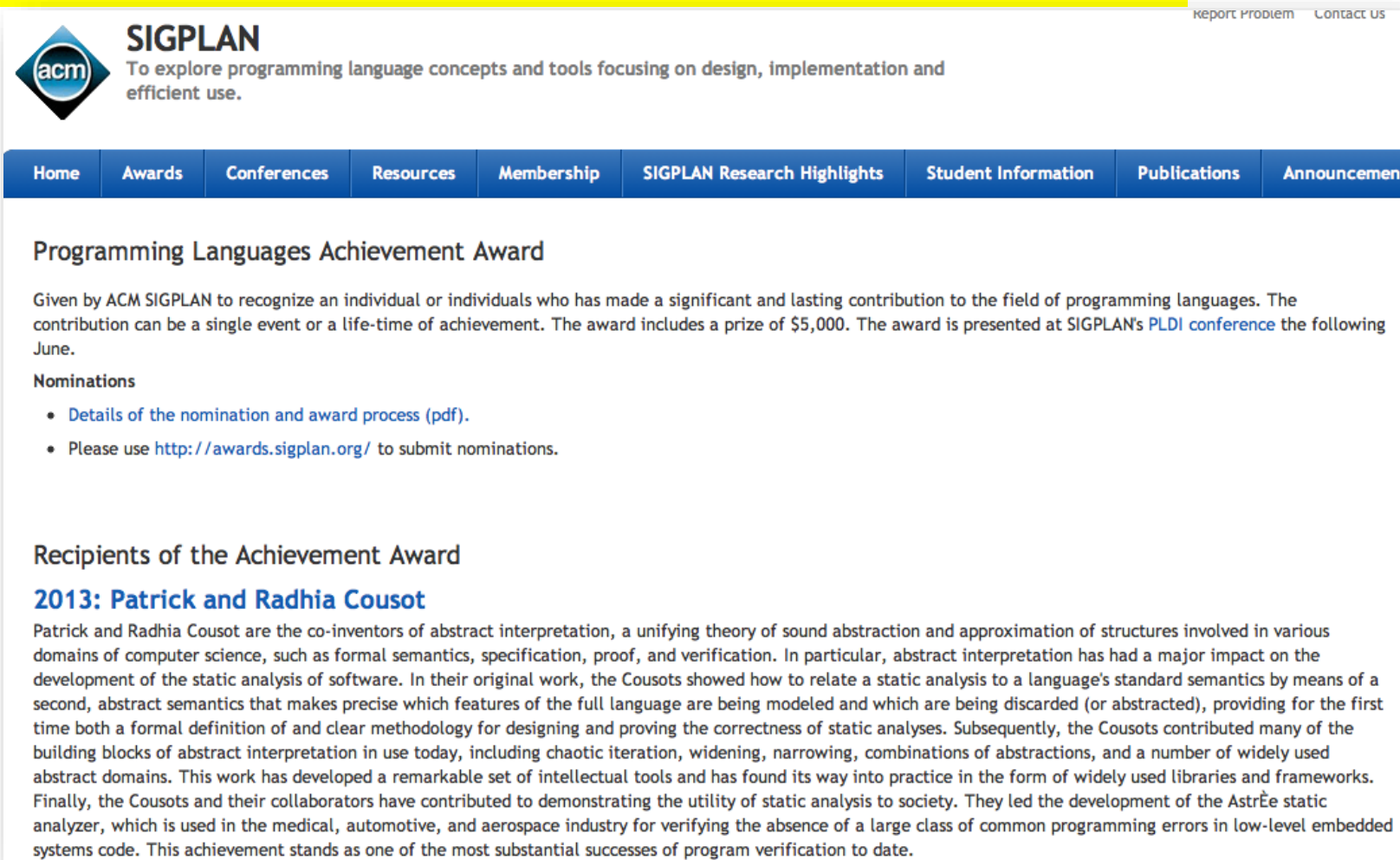
- 最早发表于POPL'77（手写的100页论文）





所获荣誉

2013年ACM SIGPLAN 程序语言成就奖



The screenshot shows the official website for the ACM SIGPLAN Programming Languages Achievement Award. At the top, there is a navigation bar with links: Home, Awards, Conferences, Resources, Membership, SIGPLAN Research Highlights, Student Information, Publications, and Announcements. The main content area is titled "Programming Languages Achievement Award" and describes the award's purpose: to recognize individuals for significant contributions to programming languages. It mentions a prize of \$5,000 and that the award is presented at the SIGPLAN PLDI conference. Below this, there is a "Nominations" section with two bullet points: "Details of the nomination and award process (pdf)." and "Please use <http://awards.sigplan.org/> to submit nominations." The "Recipients of the Achievement Award" section highlights the 2013 winners, Patrick and Radhia Cousot, and provides a detailed paragraph about their work on abstract interpretation and static analysis, noting its impact on software development and industry applications like the Astrée analyzer.

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



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](#)



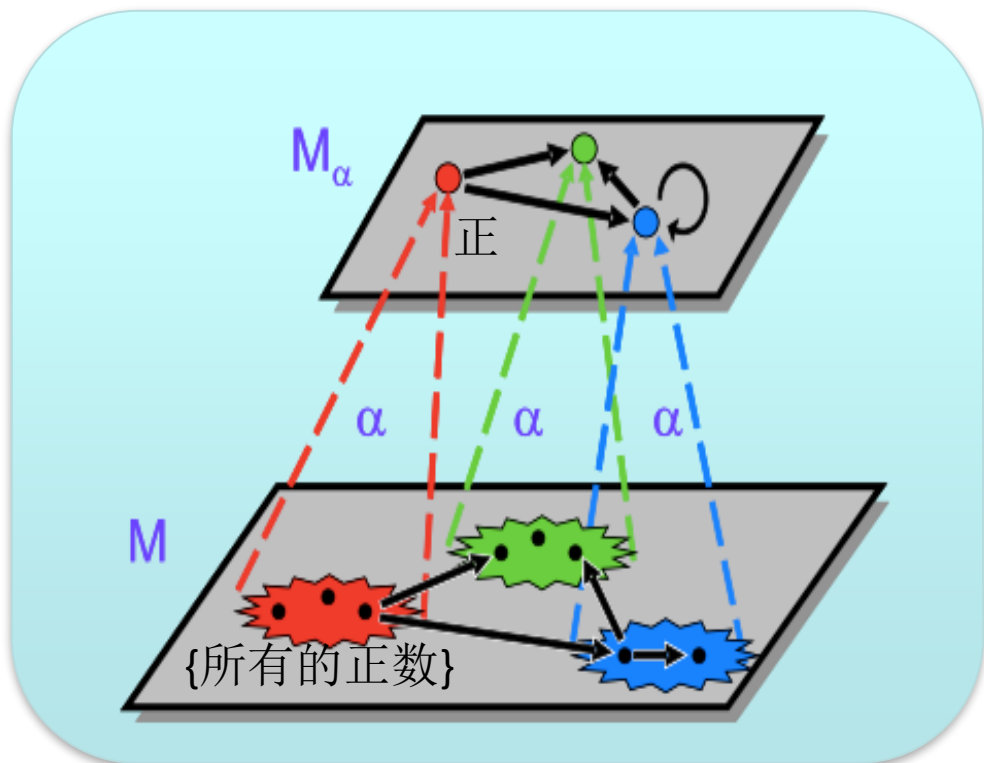
IEEE JOHN VON NEUMANN MEDAL RECIPIENTS

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值
 - $\gamma(\text{正}) = \{\text{所有的正数}\}$
 - $\gamma(\perp) = \emptyset$
 - 暂时可以把具体值想像成集合
- 抽象化函数 α 将具体值映射为抽象值
 - $\alpha(\{\text{所有的正数}\}) = \text{正}$
 - $\alpha(\{1, 2\}) = \text{正}$
 - $\alpha(\{-1, 0\}) = \text{赧}$
- 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: $\forall \text{甲} \in \text{虚}: \alpha(\gamma(\text{甲})) \sqsubseteq \text{甲}$



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



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：正常返回的执行踪迹对应变量的符号
 - 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
 - α ：踪迹集合对应的精确分析结果，定义为
 - $\alpha(X) = \sqcup_{x \in X} \beta(x)$
 - $\gamma(\text{甲}) = \{x \mid \beta(x) \sqsubseteq \text{甲}\}$

红色为特定分析需要定义的部分

- 容易证明上述元素形成伽罗瓦连接
 - $\alpha(X) \sqsubseteq \text{甲} \Rightarrow X \subseteq \gamma(\text{甲})$ ：由 γ 定义直接可得
 - $X \subseteq \gamma(\text{甲}) \Rightarrow \alpha(X) \sqsubseteq \text{甲}$ ：两边应用 α ，得 $\alpha(X) \sqsubseteq \alpha(\gamma(\text{甲})) = \sqcup_{\beta(x) \sqsubseteq \text{甲}} \beta(x) \sqsubseteq \text{甲}$
 - 最后一步用到下页最小上界定理。给定集合 $\{\beta(x) \mid \beta(x) \sqsubseteq \text{甲}\}$ ，甲是该集合的上界， $\sqcup_{\beta(x) \sqsubseteq \text{甲}} \beta(x)$ 是最小上界。



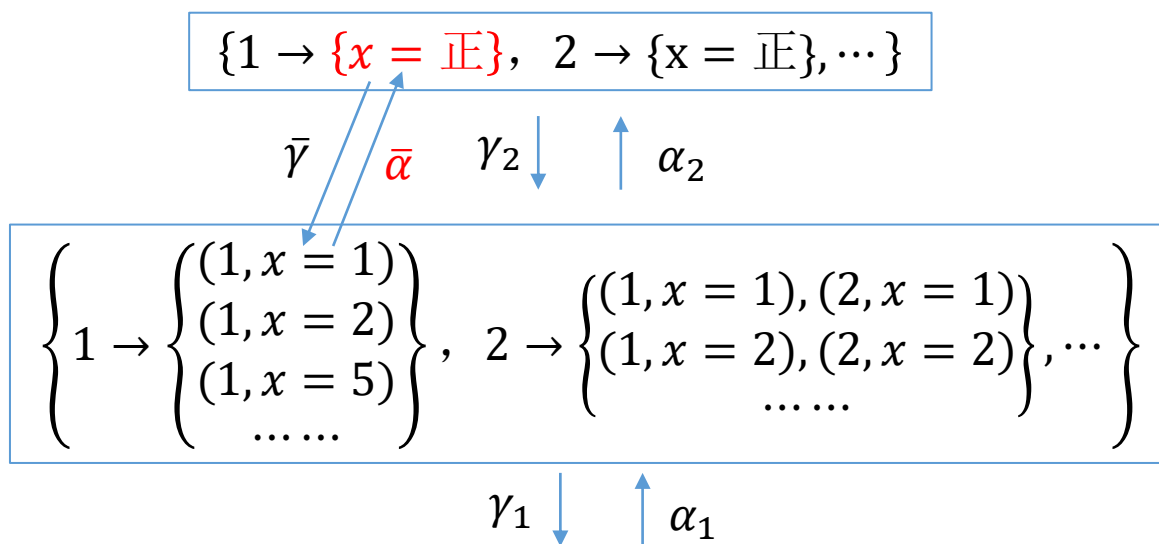
集合的最小上界

- 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 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

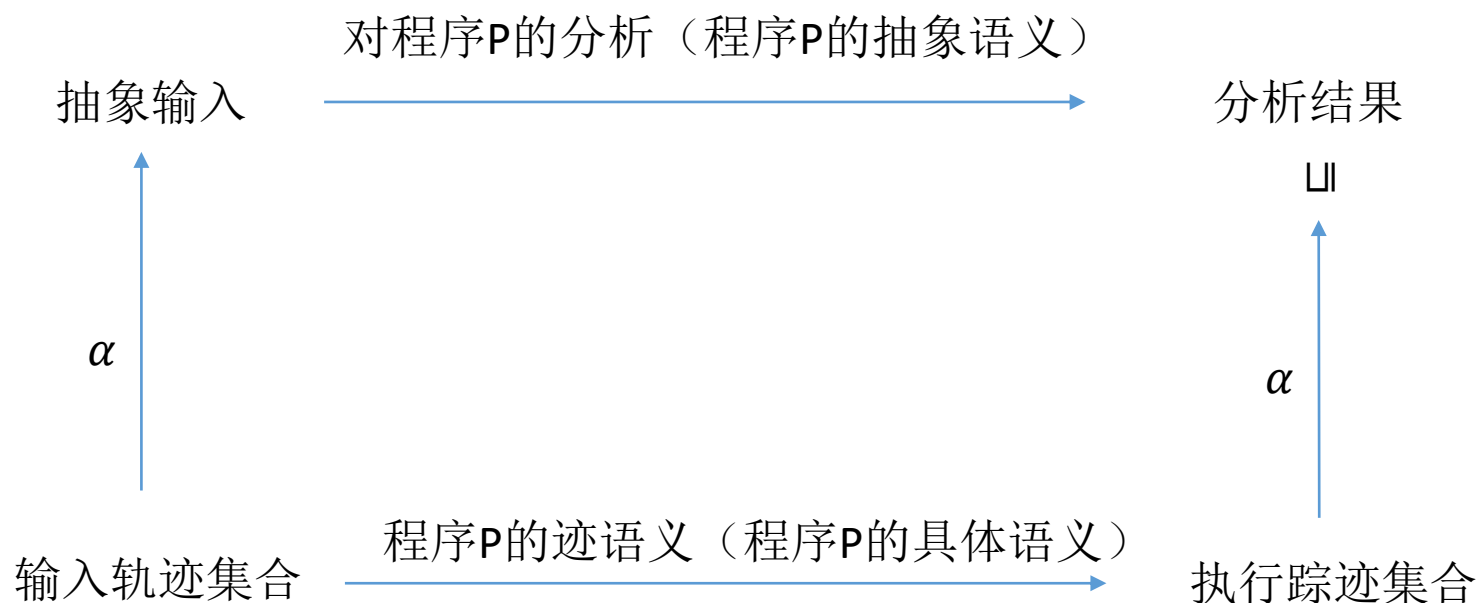


红色为需要定义的部分



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





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
 - $Step(T) = \{t + (v', m') \mid$
 $t \in T$
 $v' \in next_{last(t).node}(last(t).mem),$
 $m' \in trans_{v'}(last(t).mem)\}$
 - 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
 - $Step^\infty(T) = \lim_{n \rightarrow \infty} Step^n(T)$



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



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



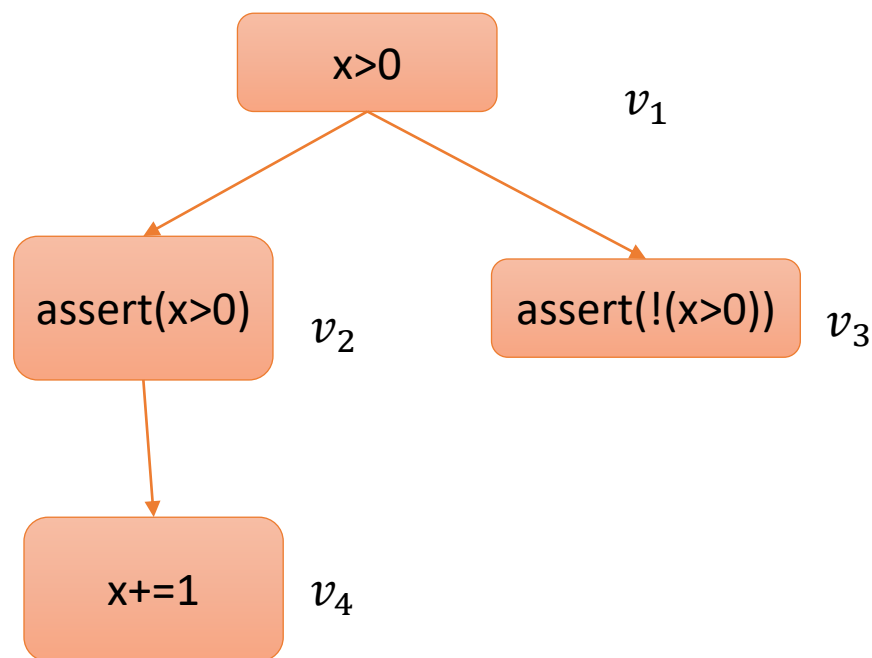
如何设计节点转换函数?

- 节点代码可能包含复杂表达式 $x := x + 1 - y$
- 如何从节点代码得到转换函数?
- 方法1: 考虑表达式求值的语义, 对应定义抽象语义并证明安全性
 - $Eval[e_1 + e_2](m) = Eval[e_1](m) + Eval[e_2](m)$
 - 符号分析 $[e_1 + e_2](\theta) = \text{符号分析}[e_1](\theta) \oplus \text{符号分析}[e_2](\theta)$
 - 可用表达式 $[e_1 + e_2] = \{e_1 + e_2\} \cup \text{可用表达式}[e_1] \cup \text{可用表达式}[e_2]$
- 方法2: 转成三地址码, 只处理每一条指令



条件压缩函数的具体语义

- $next_{v_1}(m) = \{v_2, v_3\}$
- $next_{v_2}(m) = \{v_4\}$
 - 如果m中x>0成立
- $next_{v_2}(m) = \emptyset$
 - 如果m中x>0不成立





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



如何设计条件压缩函数?

- 设计反向执行语义：给定输出的抽象值，计算输入的抽象值
 - 整数采用符号抽象，布尔值采用 $\{\perp, \text{真}, \text{假}, \text{值}\}$ ，其中 $\gamma(\text{值}) = \{\text{true}, \text{false}\}$
 - 反向 $[\wedge](\perp) = (\perp, \perp)$
 - 反向 $[\wedge](\text{真}) = (\text{真}, \text{真})$
 - 反向 $[\wedge](\text{假}) = (\text{值}, \text{值})$
 - 反向 $[\wedge](\text{值}) = (\text{值}, \text{值})$
 - 反向 $[> 0](\perp) = (\text{值})$
 - 反向 $[> 0](\text{真}) = (\text{正})$
 - 反向 $[> 0](\text{其他}) = (\text{躲})$
 - 反向 $[*](\perp) = (\perp, \perp)$
 - 反向 $[*](\text{其他}) = (\text{躲}, \text{躲})$
- 根据反向执行语义计算出变量的抽象值，然后和原来的值求交
 - 需要在抽象域上定义求交操作



更精确的反向执行语义

- 参考输入的反向执行语义：给定输入输出的抽象值，压缩输入的抽象值
 - 反向[*](甲, 甲, \perp) = (\perp , \perp)
 - 反向[*](正, 甲, 正) = (正, 甲 \sqcap 正)
 - 反向[*](负, 甲, 正) = (负, 甲 \sqcap 负)
 - 反向[*](零, 甲, 正) = (\perp , \perp)
 - 反向[*](糅, 正, 正) = (正, 正)
 -
 - 反向[>](甲, 甲, \perp) = (\perp , \perp)
 - 反向[>](负, 甲, 真) = (负, 甲 \sqcap 负)
 - 反向[>](零, 甲, 真) = (负, 甲 \sqcap 负)
 -
- 首先采用正向语义计算出表达式的值，然后再用反向语义压缩变量的值



正向反向迭代

- 反向压缩变量的值之后，再进行一次正向反向流程可能得到更精确的值
 - $x > 0 \wedge y > x \wedge z > y$
 - 假设一开始 x, y, z 的值都是罅
 - 第一轮得到 $\{x \rightarrow \text{正}, y \rightarrow \text{罅}, z \rightarrow \text{罅}\}$
 - 第二轮得到 $\{x \rightarrow \text{正}, y \rightarrow \text{正}, z \rightarrow \text{罅}\}$
 - 第三轮得到 $\{x \rightarrow \text{正}, y \rightarrow \text{正}, z \rightarrow \text{正}\}$
- 如果反向语义函数保持单调，并且确保压缩或保持输入值，同时半格高度有限，那么该迭代过程一定收敛。



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

- 不区分不同节点上的OUT值，我们就得到了流非敏感分析

- $F_{fi}(OUT) = \sqcup_{v \in V} f_v(OUT)$

- 对比流敏感分析

- $$F(OUT_{v_1}, OUT_{v_2}, \dots, OUT_{v_n}) =$$
$$\begin{aligned} & (f_{v_1}(\sqcup_{w \in \text{pred}(v_1)} OUT_w), \\ & f_{v_2}(\sqcup_{w \in \text{pred}(v_2)} OUT_w), \\ & \dots, \\ & f_{v_n}(\sqcup_{w \in \text{pred}(v_n)} OUT_w)) \end{aligned}$$

- 可以定义流非敏感结果和流敏感结果之间的伽罗瓦连接。容易看出， F_{fi} 是F的安全抽象。



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
 - 抽象域的值可以进一步映射为想要的分析结果



路径敏感性

- 路径非敏感分析：假设所有分支都可达，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
- 带条件压缩函数的分析就是路径敏感分析



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



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？

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



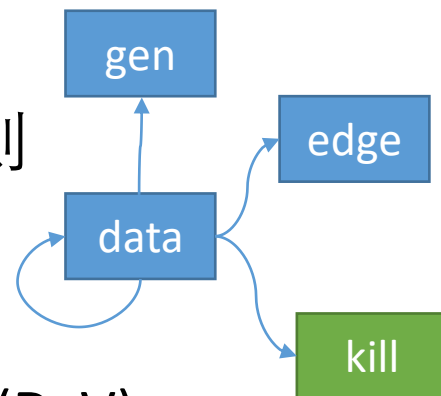
Datalog \neg

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



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



作业:

- 整数采用区间抽象, 布尔值采用{ \perp , 真, 假, 值}
- 请针对下列操作设计参考输入的反向抽象语义
 - 逻辑与
 - 逻辑非
 - 大于
 - 加法
- 要求尽可能精确
- 同时分析基于你设计的反向语义, 对任意表达式是否能保证收敛



参考资料

- 《Introduction to Static Analysis》 Rival and Yi
- Datalog Introduction
 - Jan Chomicki
 - <https://cse.buffalo.edu/~chomicki/636/datalog-h.pdf>
- Datalog引擎列表
 - <https://en.wikipedia.org/wiki/Datalog>