



软件分析

# 程序综合： 约束求解和空间表示

熊英飞

北京大学



# 约束求解法



# 约束求解法

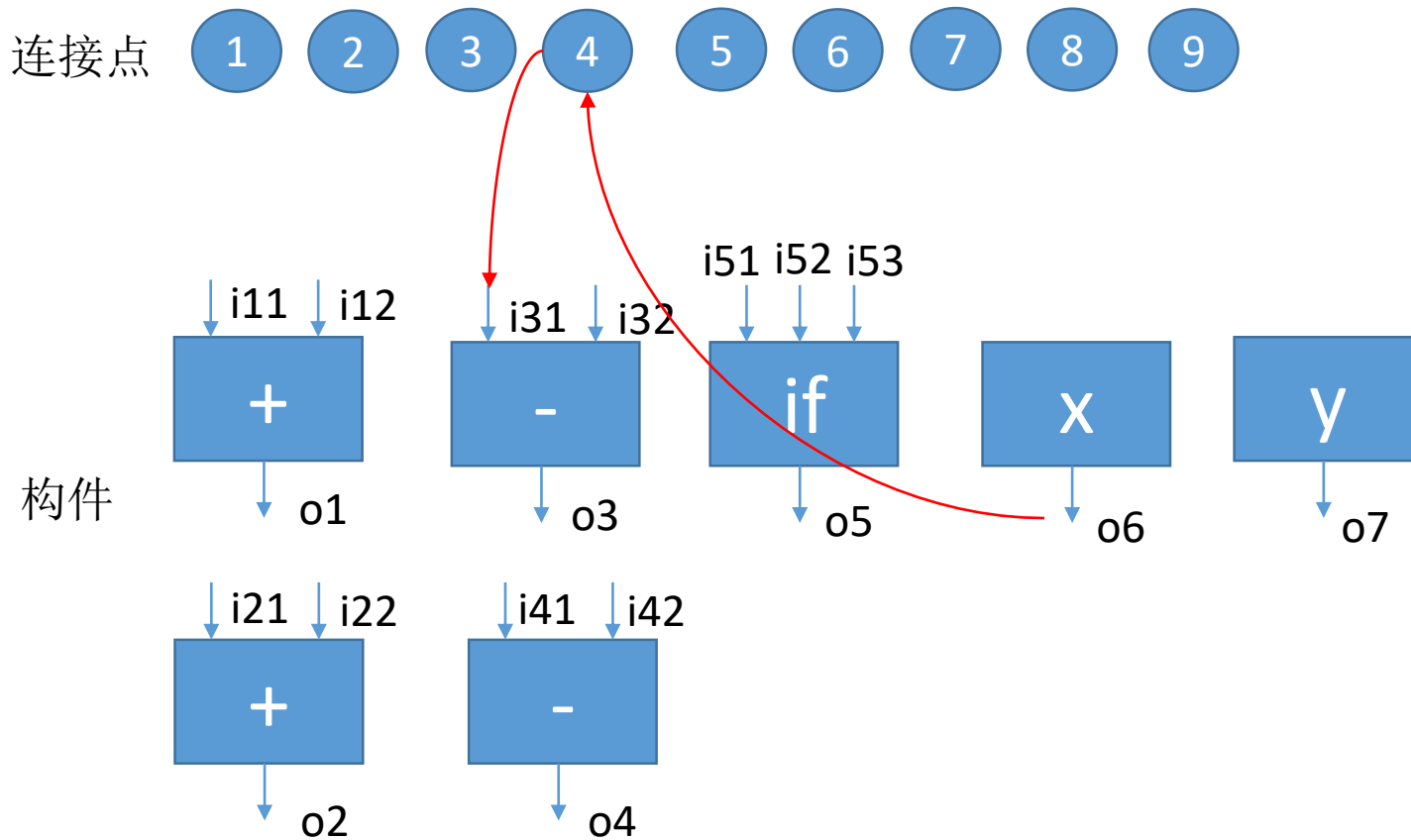
- 将程序综合问题整体转换成约束求解问题，由SMT求解器求解

# 基于构件的程序综合 Component-Based Program Synthesis



添加标签变量:

- $l_{i11}, l_{i22}, \dots$
- $l_{o1}, l_{o2}, \dots$
- $l_o$ : 程序输出



$$l_{o6} = l_{i31} = 4$$



# 产生约束

- 产生规约约束：
  - $\forall x, y: o \geq x \wedge o \geq y \wedge (o = x \vee o = y)$
- 对所有component产生语义约束：
  - $o_1 = i_{11} + i_{12}$
- 对所有的输入输出标签对产生连接约束：
  - $l_{o_1} = l_{i_{11}} \rightarrow o_1 = i_{11}$
- 对所有的输出标签产生编号范围约束
  - $l_{o_1} \geq 1 \wedge l_{o_1} \leq 9$
- 对所有的 $o_i$ 对产生唯一性约束
  - $l_{o_1} \neq l_{o_2}$
- 对统一构件的输入和输出产生防环约束
  - $l_{i_{11}} < l_{o_1}$

能否去掉连接点和输出标签 $l_{ox} \dots$ ，直接用 $l_{ixx}$ 的值表示应该连接第几号输出？



# 约束限制

- 之前的约束带有全称量词，不好求解
- 实践中通常只用于规约为输入输出样例的情况
- 假设规约为
  - $f(1,2) = 2$
  - $f(3,2) = 3$
- 则产生的约束为：
  - $x = 1 \wedge y = 2 \rightarrow o = 2$
  - $x = 3 \wedge y = 2 \rightarrow o = 3$
- 通过和CEGIS结合可以求解任意规约



# 空间表示法



# 例子：化简的max问题

- 语法：

```
Expr ::= x | y
      | Expr + Expr
      | (ite BoolExpr Expr Expr)
BoolExpr ::= BoolExpr ∧ BoolExpr
          | ¬BoolExpr
          | Expr ≤ Expr
```

- 规约：

$$\forall x, y : \mathbb{Z}, \quad \max_2(x, y) \geq x \wedge \max_2(x, y) \geq y \\ \wedge (\max_2(x, y) = x \vee \max_2(x, y) = y)$$

- 期望答案：  $\text{ite } (x \leq y) \ y \ x$





# 自顶向下遍历

- 按语法依次展开
  - Expr
  - $x, y, \text{Expr}+\text{Expr}, \text{if}(\text{BoolExpr}, \text{Expr}, \text{Expr})$
  - $y, \text{Expr}+\text{Expr}, \text{if}(\text{BoolExpr}, \text{Expr}, \text{Expr})$
  - Expr+Expr,  $\text{if}(\text{BoolExpr}, \text{Expr}, \text{Expr})$
  - $x+\text{Expr}, y+\text{Expr}, \text{Expr}+\text{Expr}+\text{Expr}$ ,  $\text{if}(\text{BoolExpr}, \text{Expr}, \text{Expr})+\text{Expr}, \text{if}(\text{BoolExpr}, \text{Expr}, \text{Expr})$
  - ...

Expr+Expr无法满足原约束  
所有展开Expr+Expr的探索都是浪费的  
如何知道这一点？



# 基于反向语义 (Inverse Semantics) 的自顶向下遍历

- 首先对规约求解或者利用CEGIS获得输入输出对
  - 求模型:  $ret \geq x \wedge ret \geq y \wedge (ret = x \vee ret = y)$
  - 得到  $x=1, y=2, ret=2$
- 由于只有加号, 任何原题目的程序都必然满足:
  - $ret \geq x \vee ret \geq y$
- 以返回值作为约束去展开该程序
  - $[2]Expr$
  - $[2]y, [1]Expr+[1]Expr, if([true]BoolExpr, [2]Expr, [*]Expr), if([false]BoolExpr, [*]Expr, [2]Expr)$
  - ...
- 只有可能满足该样例展开方式才被考虑



# Witness function

- Witness function针对反向语义具体展开分析
- 输入：
  - 样例输入，如 $\{x=1, y=2\}$
  - 期望输出上的约束，如 $[2]$ ，表示返回值等于2
  - 期望非终结符，如Expr
- 输出：
  - 一组展开式和非终结符上的约束列表，如
    - $[2]y, [1]Expr+[1]Expr, if([true]BoolExpr, [2]Expr, [*]Expr), if([false]BoolExpr, [*]Expr, [2]Expr)$
- Witness Function需要由用户提供

注：在原始文献中，witness函数细分为witness和skolemization两种函数，这里简单起见不再区分。



# Witness Function性质

- Witness Function具有必要性，如果
  - 满足原约束的所有程序都被至少一个展开式覆盖
- Witness Function具有充分性，如果
  - 满足展开式的所有程序都被原约束覆盖
- 必要的witness function保证不排除正确的程序
- 充分的witness function保证产生的程序一定是正确的



# 问题1：多样例

- 在CEGIS求解过程中，样例会逐渐增多，如何采用多个样例剪枝？



# 问题2：重复计算

- 重复计算1
  - $[1]Expr + [2]Expr$
  - 假设 $[1]Expr$ 可以展开 $n$ 个程序， $[2]Expr$ 无法展开出完整程序，但针对这 $n$ 个程序都要重复尝试展开 $[2]Expr$
- 重复计算2
  - $if([true]BoolExpr, [2]Expr, [*]Expr)$ ,
  - $if([false]BoolExpr, [*]Expr, [2]Expr)$
  - 红色和绿色部分的展开完全相同，但却分布在两颗树中



# 基于空间表示的综合

- 通过某种数据结构表示程序的集合
- 每次操作一个集合而非单个程序



# FlashMeta

- 一个基于空间表示的程序综合框架
  - 由微软的Sumit Gulwani设计
- 基本思路：
  - 采用带约束的上下文无关文法来表示程序空间，如：
    - $[2]\text{Expr} \rightarrow [2]y \mid [1]\text{Expr}+[1]\text{Expr}$
  - 对于每个样例产生一个上下文无关文法
    - 表示满足该样例的程序集合
  - 通过对上下文无关文法求交得到满足所有样例的文法



Sumit Gulwani  
14年获SIGPLAN  
Robin Milner青年  
研究者奖





# VSA

- 上下文无关语言求交之后不一定是上下文无关语言

- 反例:

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid ab$$

$$C \rightarrow cC \mid c$$

$$S' \rightarrow A'C'$$

$$A' \rightarrow aA' \mid a$$

$$C' \rightarrow bC'c \mid bc$$

$S \cap S'$ 不是上下文无关语言

- FlashMeta采用了VSA来表示程序子空间
  - Version Space Algebra(VSA)是上下文无关文法的子集
  - VSA求交一定是VSA



# VSA

- VSA是只包含如下三种形式的上下文无关文法，且每个非终结符只在左边出现一次
  - $N \rightarrow p_1 \mid p_2 \mid \dots \mid p_n$
  - $N \rightarrow N_1 \mid N_2 \mid \dots \mid N_n$
  - $N \rightarrow f(N_1, N_2, \dots, N_n)$
  - $N$ 是非终结符， $p$ 是终结符列表， $f$ 是终结符
- 无递归时，VSA可表示产生式数量指数级的程序空间。
- 有递归时，VSA可表示无限大的程序空间。

# VSA例子

```
Expr ::= x | y
      | Expr + Expr
      | (ite BoolExpr Expr Expr)
BoolExpr ::= BoolExpr ^ BoolExpr
          | ¬BoolExpr
          | Expr ≤ Expr
```

- $\text{Expr} ::= V \mid \text{Add} \mid \text{If}$
- $\text{Add} ::= + (\text{Expr}, \text{Expr})$
- $\text{If} ::= \text{ite}(\text{BoolExpr}, \text{Expr}, \text{Expr})$   
 $V ::= x \mid y$
- $\text{BoolExpr} ::= \text{And} \mid \text{Neg} \mid \text{Less}$
- $\text{And} ::= \wedge(\text{BoolExpr}, \text{BoolExpr})$
- $\text{Neg} ::= \text{Not}(\text{BoolExpr})$
- $\text{Less} ::= \leq(\text{Expr}, \text{Expr})$



# 无法表示成VSA的例子

- 无法表示成VSA的上下文无关文法的例子

$$A \rightarrow aAb \mid ab$$

$$C \rightarrow cC \mid c$$

- 即：VSA通过括号确定了语法树的结构，只能采用固定方式解析



# 自顶向下构造VSA

- 给定输入输出样例，递归调用witness function，将约束和原非终结符同时作为新非终结符
- $[2]Expr \rightarrow y \mid [1]Expr + [1]Expr \mid$   
 $if([true]BoolExpr)[2]Expr [*]Expr \mid$   
 $if([false]BoolExpr)...$
- $[1]Expr \rightarrow x$
- $[*]Expr \rightarrow ...$
- $[true]BoolExpr \rightarrow true \mid \neg [false]BoolExpr \mid [2]Expr \leq [2]$   
 $Expr \mid [1]Expr \leq [2]Expr \mid [1]Expr \leq [1]Expr \mid ...$



# 自顶向下构造VSA

- 根据witness function的实现，有可能出现非终结符无法展开的情况
  - VSA生成后，递归删除所有展开式为空的非终结符
  - 假设 $x=y=2$
  - ~~$[3]Expr \rightarrow [2]Expr + [1]Expr \mid [1]Expr + [2]Expr$~~
  - ~~$[2]Expr \rightarrow x \mid y$~~
  - ~~$[1]Expr \rightarrow c$~~
- While(有非终结符展开为空) {  
    删除该非终结符  
    删除所有包含该非终结符的产生式  
}
- 删除所有不在右边出现的非终结符



# VSA求交

- $intersect(N, N')$  输出把非终结符  $N$  和  $N'$  求交之后的产生式
- $[N \cap N']$  表示把  $N$  和  $N'$  求交之后的终结符
- 如果  $N \rightarrow N_1 \mid N_2 \mid \dots$ 
  - $intersect(N, N') = \{ [N \cap N'] \rightarrow [N_1 \cap N'] \mid [N_2 \cap N'] \mid \dots \} \cup intersect(N_1, N') \cup intersect(N_2, N') \cup \dots$
- 如果  $N \rightarrow f(N_1 \mid \dots \mid N_k)$  且  $N' \rightarrow f'(N'_1 \mid \dots \mid N'_{k'})$  且  $f \neq f'$  或者  $k \neq k'$ 
  - $intersect(N, N') = \{ [N \cap N'] \rightarrow \epsilon \}$
- 如果  $N \rightarrow f(N_1 \mid \dots \mid N_k)$  且  $N' \rightarrow f(N'_1 \mid \dots \mid N'_k)$ 
  - $intersect(N, N') = \{ [N \cap N'] \rightarrow f([N_1 \cap N'_1], \dots, [N_k \cap N'_k]) \} \cup intersect(N_1, N'_1) \cup \dots \cup intersect(N_k, N'_k)$



# VSA求交

- 如果  $N \rightarrow f(N_1 \mid \dots \mid N_k)$  且  $N' \rightarrow p'_1 \mid p'_2 \mid \dots$  且存在  $p'_{j_1}, \dots, p'_{j_m}$  使得  $N$  可以生成  $p'_{j_1}, \dots, p'_{j_m}$  中的每一个而不能生成别的
  - $intersect(N, N') = \{[N \cap N'] \rightarrow p'_{j_1}, \dots, [N \cap N'] \rightarrow p'_{j_m}\}$
  - 否则  $intersect(N, N') = \{[N \cap N'] \rightarrow \epsilon\}$
- 如果  $N \rightarrow p_1 \mid p_2 \mid \dots$  且  $N' \rightarrow p'_1 \mid p'_2 \mid \dots$  且  $p''_{j_1}, \dots, p''_{j_m}$  是两者的公共部分
  - $intersect(N, N') = \{[N \cap N'] \rightarrow p''_{j_1}, \dots, [N \cap N'] \rightarrow p''_{j_m}\}$
- 如果不符合以上情况， 则
  - $intersect(N, N') = intersect(N', N)$
- 最后再删掉所有产生结果都包括  $\perp$  的非终结符





# 完整FlashMeta的例子

- 考虑字符串拼接
- 语法：
  - $S \rightarrow S + S \mid x \mid y \mid z$
- 例子1：
  - $ret = \text{"acc"}$
  - $x = \text{"a"}$
  - $y = \text{"cc"}$
  - $z = \text{"c"}$

生成VSA:

- $[acc]S \rightarrow [a]S + [cc]S$   
 $\mid [ac]S + [c]S$
- $[ac]S \rightarrow [a]S + [c]S$
- $[cc]S \rightarrow [c]S + [c]S \mid y$
- $[a]S \rightarrow x$
- $[c]S \rightarrow z$

简单起见，不严格采用VSA语法



# 完整FlashMeta的例子

- 考虑字符串拼接

- 语法:

- $S \rightarrow S + S \mid x \mid y \mid z$

- 例子1:

- $ret = \text{"aac"}$
  - $x = \text{"a"}$
  - $y = \text{"ac"}$
  - $z = \text{"c"}$

生成VSA:

- $[aac]S \rightarrow [a]S + [ac]S$   
|  $[aa]S + [c]S$
- $[ac]S \rightarrow [a]S + [c]S \mid y$
- $[aa]S \rightarrow [a]S + [a]S$
- $[a]S \rightarrow x$
- $[c]S \rightarrow z$



# VSA求交

$[acc]S \rightarrow [a]S + [cc]S \mid [ac]S + [c]S$   
 $[ac]S \rightarrow [a]S + [c]S$   
 $[cc]S \rightarrow [c]S + [c]S \mid y$   
 $[a]S \rightarrow x$   
 $[c]S \rightarrow z$



$[aac]S \rightarrow [a]S + [ac]S \mid [aa]S + [c]S$   
 $[ac]S \rightarrow [a]S + [c]S \mid y$   
 $[aa]S \rightarrow [a]S + [a]S$   
 $[a]S \rightarrow x$   
 $[c]S \rightarrow z$

==

~~$[acc, aac]S \rightarrow [a, a]S + [cc, ac]S \mid [a, aa]S + [cc, c]S \mid [ac, a]S + [c, ac]S$~~   
 ~~$[ac, aa]S + [c, c]S$~~   
 $[a, a]S \rightarrow x$   
 ~~$[c, c]S \rightarrow z$~~   
 $[cc, ac]S \rightarrow [c, a]S + [c, c]S \mid y$   
 ~~$[a, aa]S \rightarrow \epsilon$~~   
 ~~$[cc, c]S \rightarrow \epsilon$~~   
 ~~$[ac, a]S \rightarrow \epsilon$~~   
 ~~$[c, ac]S \rightarrow \epsilon$~~   
 ~~$[ac, aa]S \rightarrow [a, a]S + [c, a]S$~~   
 ~~$[c, a]S \rightarrow \epsilon$~~



# 问题回顾

- 在CEGIS求解过程中，样例会逐渐增多，如何采用多个样例剪枝？
  - FlashMeta通过VSA求交解决多样例问题
- 重复计算1
  - [1]Expr+[2]Expr
  - 假设[1]Expr可以展开n个程序，[2]Expr无法展开出完整程序，但针对这n个程序都要重复尝试展开[2]Expr
  - FlashMeta通过分治，对两个子问题分别处理
- 重复计算2
  - if([true]BoolExpr, [2]Expr, [\*]Expr),
  - if([false]BoolExpr, [\*]Expr, [2]Expr)
  - 红色和绿色部分的展开完全相同，但却分布在两颗树中
  - FlashMeta通过动态规划，对相同的子问题复用



# 自底向上构造VSA

- Witness Function需要手动撰写，且撰写良好的Witness Function并不容易
- 解决思路：
  - 利用程序操作符本身的语义自底向上构造VSA，避免反向语义
  - 也被称为基于Finite Tree Automata (FTA) 的方法



# 自底向上构造VSA

- 维护一个非终结符集合和产生式集合
- 初试非终结符包括输入变量:  $[2]x, [1]y$
- 反复用原产生式匹配非终结符, 得到新产生式和新的非终结符。
- 重复上述过程直到得到起始符号和期望输出

## 非终结符集合

$[2]x$   
 $[1]y$   
 $[2]Expr$   
 $[1]Expr$   
 $[3]Expr$

$Expr \rightarrow x$   
 $Expr \rightarrow y$   
 $Expr \rightarrow Expr + Expr$

## 产生式集合

$[2]Expr \rightarrow [2]x$   
 $[1]Expr \rightarrow [1]y$   
 $[3]Expr \rightarrow [2]Expr + [1]Expr$



# 自底向上vs自顶向下

- 两种方法有不同的适用范围
  - 自顶向下适用于从输出出发选项较少的情况
    - 如：字符串拼接
  - 自底向上适用于从输入出发选项较少的情况
    - 如：实数运算



# 基于抽象精化的合成





# 例子

- $n \rightarrow x \mid n + t \mid n \times t$
- $t \rightarrow 2 \mid 3$
- 输入：  $x=1$ ， 输出：  $ret=9$
- 目标程序举例：  $(x+2)*3$
  
- 按某通用witness函数分解得到
- $[9]n_1 \rightarrow [1]n + [8]t \mid [2]n + [7]t \mid \dots$   
 $\mid [1]n \times [9]t \mid [3]n \times [3]t \mid [9]n \times [1]t$

大量展开式都是无效的  
能否一次排除而不是一个一个排除？



# 基本思想

- 之前见到的VSA按具体执行结果组织程序
- 但对于特定规约，很多具体程序是等价的
- 按抽象域组织程序可以进一步合并同类项
  
- 即：
- $[[5,12]]n \rightarrow [[0,4]]n + [[5,8]]t$
  
- 如何知道适合当前规约的抽象域是什么？
  - 从最抽象的抽象域开始，逐步精华



# 元抽象域

- 元抽象域由一组抽象值的集合构成，如
  - 蹀，即  $x \in [-\infty, +\infty]$
  - ...  $-7 \leq x \leq 0, 1 \leq x \leq 8, 9 \leq x \leq 18, \dots$
  - ...  $-3 \leq x \leq 0, 1 \leq x \leq 4, 5 \leq x \leq 8, \dots$
  - ...  $-1 \leq x \leq 0, 1 \leq x \leq 2, 3 \leq x \leq 4, \dots$
  - ...  $x = -1, x = 0, x = 1, \dots$
- 要求：
  - 包括蹀，且  $\gamma(\text{蹀}) = \text{具体值的全集}$
  - 包括所有的具体值，且对任意具体值  $a$ ， $\alpha(\{a\}) = a \wedge \gamma(a) = \{a\}$
  - 对元抽象域的任意子集可以定义封闭的抽象运算
- 实际抽象域的抽象值由元抽象域的值构成
- 一开始只包含蹀，在精化过程中逐步增加



# 1.1 抽象域上的计算

- 抽象域包括 罅
- 自底向上构造VSA，得
  - $[\text{罅}]n \rightarrow [\text{罅}]x \mid [\text{罅}]n + [\text{罅}]t \mid [\text{罅}]n \times [\text{罅}]t$
  - $[\text{罅}]t \rightarrow [\text{罅}]2 \mid [\text{罅}]3$
- 输入为 $x=\text{罅}$ ，输出为 $\text{ret}=\text{罅}$
- 随机从VSA中采样程序，得到 $\text{ret}=x$



# 1.2 抽象域的精化

查找一个极大的抽象值，包含计算值但不包含期望值  
添加抽象值[1, 8]

期望值	棵	9	棵
计算值	x:棵	x:1	x:[1,8]
	抽象域计算	实际域上反例	精华后的抽象域计算

精化后抽象域的性质:

抽象域的运算结果一定包括反例输入在具体域上的运算结果

抽象域的运算结果一定不包括反例的期望输出



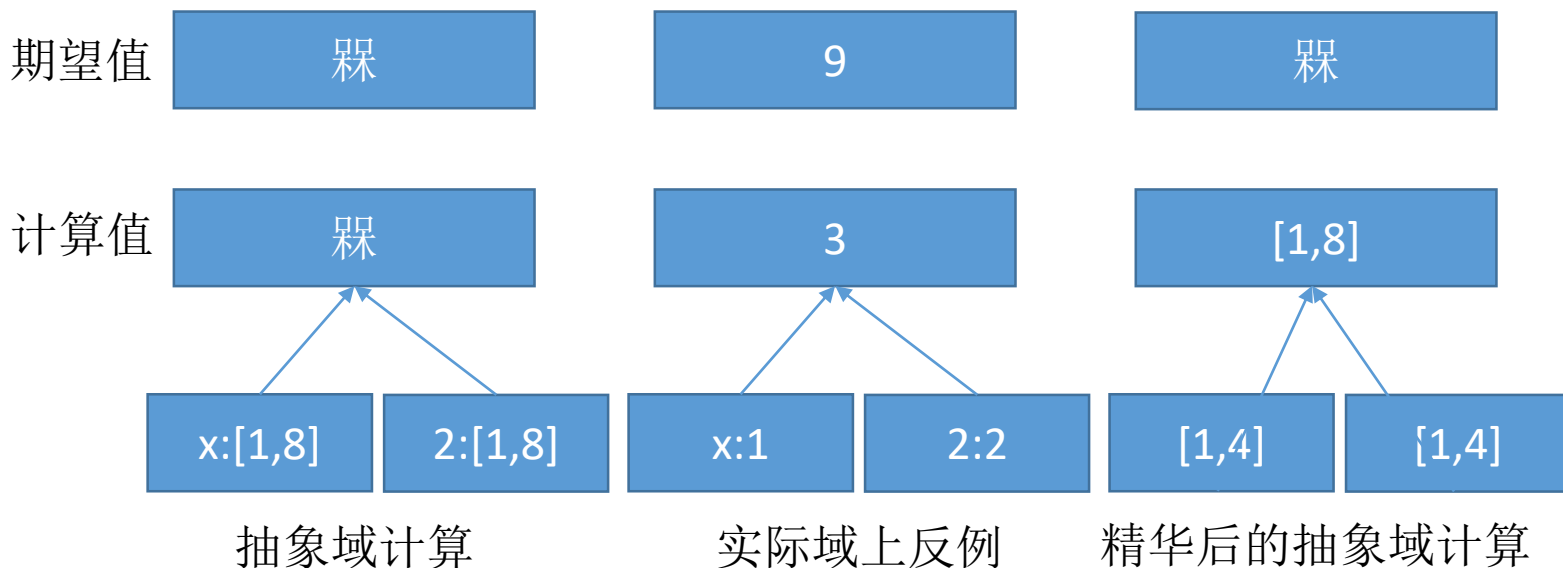
## 2.1 抽象域上的计算

- 抽象域包括 { $\text{nil}$ ,  $[1, 8]$ }
- 自底向上构造VSA, 得
  - $[\text{nil}]n \rightarrow [\text{nil}]n + [1, 8]t \mid [\text{nil}]n \times [1, 8]t \mid [1, 8]n + [1, 8]t \mid [1, 8]n \times [1, 8]t$
  - $[1, 8]n \rightarrow [1, 8]x$
  - $[1, 8]t \rightarrow [1, 8]2 \mid [1, 8]3$
- 输入为 $x=[1, 8]$ , 输出为 $\text{ret}=\text{nil}$
- 随机从VSA中采样程序, 得到 $\text{ret}=x+2$



## 2.2 抽象域的精化

- 自顶向下依次精化每个节点
  - 如果孩子节点在抽象域上计算结果不等于当前结点的抽象值
    - 对孩子列表寻找一个极大的抽象值列表，使得该抽象值列表覆盖计算值，且抽象域上计算结果 $\sqsubseteq$ 当前结点抽象值
  - 添加抽象值[1, 4]



精化后抽象域的性质：

抽象域的运算结果一定包括反例输入在具体域上的运算结果

抽象域的运算结果一定不包括反例的期望输出



## 3.1 抽象域上的计算

- 抽象域包括  $\{\text{靛}, [1, 8], [1, 4]\}$
- 自底向上构造VSA, 得
  - $[ \text{靛} ] n \rightarrow [ \text{靛} ] n + [1, 4] t \mid [ \text{靛} ] n \times [1, 4] t \mid [1, 8] n + [1, 4] t \mid [1, 8] n \times [1, 4] t \mid \dots$
  - $[1, 8] n \rightarrow [1, 4] n + [1, 4] t \mid \dots$
  - $[1, 4] n \rightarrow x$
  - $[1, 4] t \rightarrow 2 \mid 3$
- 输入为 $x=[1, 4]$ , 输出为 $\text{ret}=\text{靛}$
- 随机从VSA中采样程序, 得到 $\text{ret}=(x+2)*3$





# 计算过程的性质

- 给定反例 $e$ 和精化后的抽象域**虚**，则
- **虚**上的运算结果一定包括反例输入在具体域上的运算结果
  - 根据安全抽象的定义可得
- **虚**上的运算结果一定不包括反例的期望输出
  - 因为第一步找到的输出不包含具体值
- 精化过程的每一步一定能找到相应抽象值
  - 因为最坏情况可以加具体值
- 即使最后的VSA也比完整的VSA小很多，实现加速



# 参考文献

- Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, Ashish Tiwari: Oracle-guided component-based program synthesis. ICSE (1) 2010: 215-224
- Polozov O , Gulwani S . FlashMeta: a framework for inductive program synthesis[C]// Acm Sigplan International Conference on Object-oriented Programming. ACM, 2015.
- Xinyu Wang, Isil Dillig, and Rishabh Singh。 Synthesis of Data Completion Scripts using Finite Tree Automata. OOPSLA, 2017
- Wang X , Dillig I , Singh R . Program Synthesis using Abstraction Refinement[J]. POPL 2018.