软件分析

程序合成：空间表示和约束求解

熊英飞
北京大学

反向语义和动态规划

例子：化简的max问题

- 语法：

$$
\begin{array}{lll}
\texttt{Expr} & ::= & x \quad | \quad y \\
 & | & \texttt{Expr} + \texttt{Expr} \\
 & | & (\texttt{ite BoolExpr Expr Expr}) \\
\texttt{BoolExpr} & ::= & \texttt{BoolExpr} \wedge \texttt{BoolExpr} \\
 & | & \neg\texttt{BoolExpr} \\
 & | & \texttt{Expr} \leq \texttt{Expr}
\end{array}
$$

- 规约：

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

- 期望答案：ite (x <= y) y x

搜索过程的冗余

- 考虑当前搜索想要满足样例(x=1, y=2)->(ret=2)
- 假设自顶向下遍历找到了这样的表达式
  - ite BoolExpr Expr1 Expr2
- 如果BoolExpr返回True，那么Expr1应该返回2
  - 对于所有返回True的BoolExpr，都需要寻找返回2的Expr1，但每次都重新寻找
- 如果BoolExpr返回False，那么Expr2应该返回2
  - “寻找返回2的Expr1”和“寻找返回2的Expr2”是完全一样的问题，但每次都重新寻找
- 以上问题都是重复计算了子问题
  - 动态规划：记录并重用求解过的子问题
  - 如何表示和分解子问题?

基于反向语义（Inverse Semantics）的自顶向下遍历

- 表示子问题：[返回值]非终结符
  - [2]Expr：寻找在当前样例上返回2的以非终结符Expr展开的表达式
  - [*]Expr：寻找任意返回值的以非终结符Expr展开的表达式
- 分解子问题：基于反向语义
  - [2]Expr
  - [2]x, [2]y, [0]Expr+[2]Expr, [1]Expr+[1]Expr, [2]Expr+[0]Expr, if([true]BoolExpr, [2]Expr, [*]Expr), if([false]BoolExpr, [*]Expr, [2]Expr)
- 之后根据需要分别求解子问题
  - [2]x, [2]y, [0]Expr, [1]Expr, [2]Expr, [true]BoolExpr, [false]BoolExpr, [*]Expr
  - 如果遇到重复的子问题就重用

Witness Function

- 一般把反向语义和剪枝合并定义为Witness function
- 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)
    - 注意样例上无解的子问题已经被剪枝

- Witness Function需要由用户提供
- 但针对每个DSL只需要提供一次

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

伪码

```
search([o]N) {
  if ([o]N求解过) 返回记录的解;
  if (N是变量)
    if(o和输入一致) 返回N;
    else 返回无解;
  options=witness([o]N);
  foreach(option in options) {
    递归求解option中的子问题;
    if(任意子问题无解) continue;
    else 根据option组合出最终程序并记录返回; }
  返回无解; }
```

多样例的情况

- 样例1：(x=1, y=2)->(ret=2)
- 样例2：(x=3, y=3)->(ret=3)
- 子问题：[2, 3]Expr
  - 在样例1上返回2，在样例2上返回3
- Witness Function同时考虑多个样例即可
  - 输入：[2, 3]Expr
  - 输出：[2, 3]y,
    if([true, true]BoolExpr, [2,3]Expr, [*]Expr),
    if([true, false]BoolExpr, [2,*]Expr, [*,3]Expr),
    if([false, true]BoolExpr, [*,3]Expr, [2,*]Expr),
    if([false, false]BoolExpr, [*,*]Expr, [2,3]Expr)

双向搜索

- 自顶向下遍历不断产生新的需要满足的输出
- 自底向上遍历不断产生新的需要满足的输入
- 也可以将子问题定义为输入输出的映射
- 子问题：[1]x, [2]y -> [2]Expr
  - 输入为x=1, y=2时，返回2的表达式
- 同时从输入输出出发进行搜索
  - 由于子问题较难被复用，在表达式合成中通常效率不如单独自顶向下或单独自底向上
- 通常用于pipeline程序或者有副作用的程序
  - 如：汇编语言的合成
  - Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, Dinakar Dhurjati: Scaling up Superoptimization. ASPLOS 2016. 297-310

空间表示法

多样例的影响

- 样例越多，子问题的数量就越多
  - 子问题数量随样例数量指数增长
- 复用子问题的可能性就越小
- 如何能增大复用子问题的机会?

基于空间表示的合成

- 通过某种数据结构表示程序的集合
- 对于每个样例产生一个程序的集合
- 对于所有集合求交得到最终程序的集合
  - 数据结构需要能支持求交操作

FlashMeta

- 一个基于空间表示的程序合成框架
  - 由微软的Sumit Gulwanid等人设计
- 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]Expr → [2]y | [1]Expr+[1]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
微软研究院研究员
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：

$$S \to AC$$
$$A \to aAb \mid ab$$
$$C \to cC \mid c$$

$$S' \to A'C'$$
$$A' \to aA' \mid a$$
$$C' \to bC'c \mid bc$$

$S \cap S'$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，且每个非终结只在左边出现一次
  - $N \rightarrow p_1 \mid p_2 \mid \cdots \mid p_n$
  - $N \rightarrow N_1 \mid N_2 \mid \cdots \mid N_n$
  - $N \rightarrow f(N_1, N_2, \ldots, N_n)$
  - N是非终结符，p是终结符列表，f 是终结符

VSA例子

$$
\begin{array}{lcl}
\texttt{Expr} & ::= & x \quad | \quad y \\
 & | & \texttt{Expr} + \texttt{Expr} \\
 & | & (\texttt{ite BoolExpr Expr Expr}) \\
\texttt{BoolExpr} & ::= & \texttt{BoolExpr} \wedge \texttt{BoolExpr} \\
 & | & \neg\texttt{BoolExpr} \\
 & | & \texttt{Expr} \leq \texttt{Expr}
\end{array}
$$

- Expr ::= V | Add | If
- Add ::= + (Expr, Expr)
- If ::= ite(BoolExpr, Expr, Expr)
  V ::= x | y
- BoolExpr ::= And | Neg | Less
- And ::= ∧(BoolExpr, BoolExpr)
- Neg ::= Not(BoolExpr)
- Less ::= <=(Expr, 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

$$S \to AC$$
$$A \to aAb \mid ab$$
$$C \to cC \mid c$$

- 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→y | [1]Expr+[1]Expr | if([true]BoolExpr)[2]Expr [∗]Expr | if([false]BoolExpr)…
- [1]Expr→x
- [∗]Expr→…
- [true]BoolExpr→true | ¬[false]BoolExpr | [2]Expr≤[2]Expr | [1]Expr≤[2]Expr | [1]Expr≤[1]Expr | …

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→[2]Expr+[1]~~E~~xpr | [1]Expr~~+[2]Expr
- ~~[2]Expr→x | y~~
- ~~[1]Expr→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

- $[N \cap N']$表示把N和N'求交之后的终结符
- 如果$N \to N_1 \mid N_2 \mid \cdots$
  - $[N \cap N'] \to [N_1 \cap N'] \mid [N_2 \cap N'] \mid \cdots$
- 如果$N \to f(N_1 \mid \ldots \mid N_k)$且$N' \to f'(N'_1 \mid \ldots \mid N'_{k'})$且$f \neq f'$或者$k \neq k'$
  - $[N \cap N'] \to \epsilon$
- 如果$N \to f(N_1 \mid \ldots \mid N_k)$且$N' \to f(N'_1 \mid \ldots \mid N'_k)$
  - $[N \cap N'] \to f([N_1 \cap N'_1], \ldots [N_k \cap N'_k])$

VSA求交

- 如果$N \to p_1 \mid p_2 \mid \cdots$，则将$N'$全部展开，和$\{p_1, p_2, \ldots\}$求交得到$\{p'_{j1}, p'_{j2}, \ldots\}$
  - $[N \cap N'] \to p'_{j1} \mid p'_{j2} \mid \cdots$
- 注意$[N \cap N']$等价于$[N' \cap N]$，所以以上规则覆盖了所有情况

完整FlashMeta的例子

- 考虑字符串拼接
- 语法：
  - S-> S + S | x | y | z
- 例子1：
  - ret="acc"
  - x="a"
  - y="cc"
  - z="c"

生成VSA：

- [acc]S->[a]S+[cc]S | [ac]S+[c]S
- [ac]S->[a]S+[c]S
- [cc]S->[c]S+[c]S | y
- [a]S→x
- [c]S→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-> S + S | x | y | z
- 例子1：
  - ret=“aac”
  - x=“a”
  - y=“ac”
  - z=“c”

生成VSA：

- [aac]S->[a]S+[ac]S | [aa]S+[c]S
- [ac]S->[a]S+[c]S | y
- [aa]S->[a]S+[a]S
- [a]S→x
- [c]S→z

VSA求交

[acc]S->[a]S+[cc]S | [ac]S+[c]S
[ac]S->[a]S+[c]S
[cc]S->[c]S+[c]S | y
[a]S→x
[c]S→z

∩

[aac]S->[a]S+[ac]S | [aa]S+[c]S
[ac]S->[a]S+[c]S | y
[aa]S->[a]S+[a]S
[a]S→x
[c]S→z

=

[acc,aac]S -> [a,a]S+[cc,ac]S | ~~[a,aa]S+[cc,c]S~~ | ~~[ac,a]S+[c,ac]S~~ |
~~[ac,aa]S+[c,c]S~~
[a,a]S -> x
~~[c,c]S -> z~~
[cc,ac]S -> ~~[c, a]S+[c,c]S |~~ y
~~[a,aa]S -> ϵ~~
~~[cc,c]S -> ϵ~~
~~[ac,a]S -> ϵ~~
~~[c,ac]S -> ϵ~~
~~[ac,aa]S -> [a,a]S+[c,a]S~~
~~[c, a]S -> ϵ~~

多样例直接构造 vs 单样例分别构造并求交

- VSA也可以通过多个样例直接构造
  - 类似基于反向语义的自顶向下搜索
- 多个样例直接构造：
  - 优势：可以利用多个样例同时剪枝
  - 劣势：子问题大幅增多，复用困难
- 通常劣势>>优势，所以FlashMeta采用了单样例分别构造并求交的方式
  - 但二者之间的权衡仍然需要进一步研究

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

王新宇
密西根大学
助理教授

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→x

Expr→ y

Expr→Expr+Expr

产生式集合

[2]Expr→[2]x

[1]Expr→[1]y

[3]Expr→[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

约束求解法

约束求解法

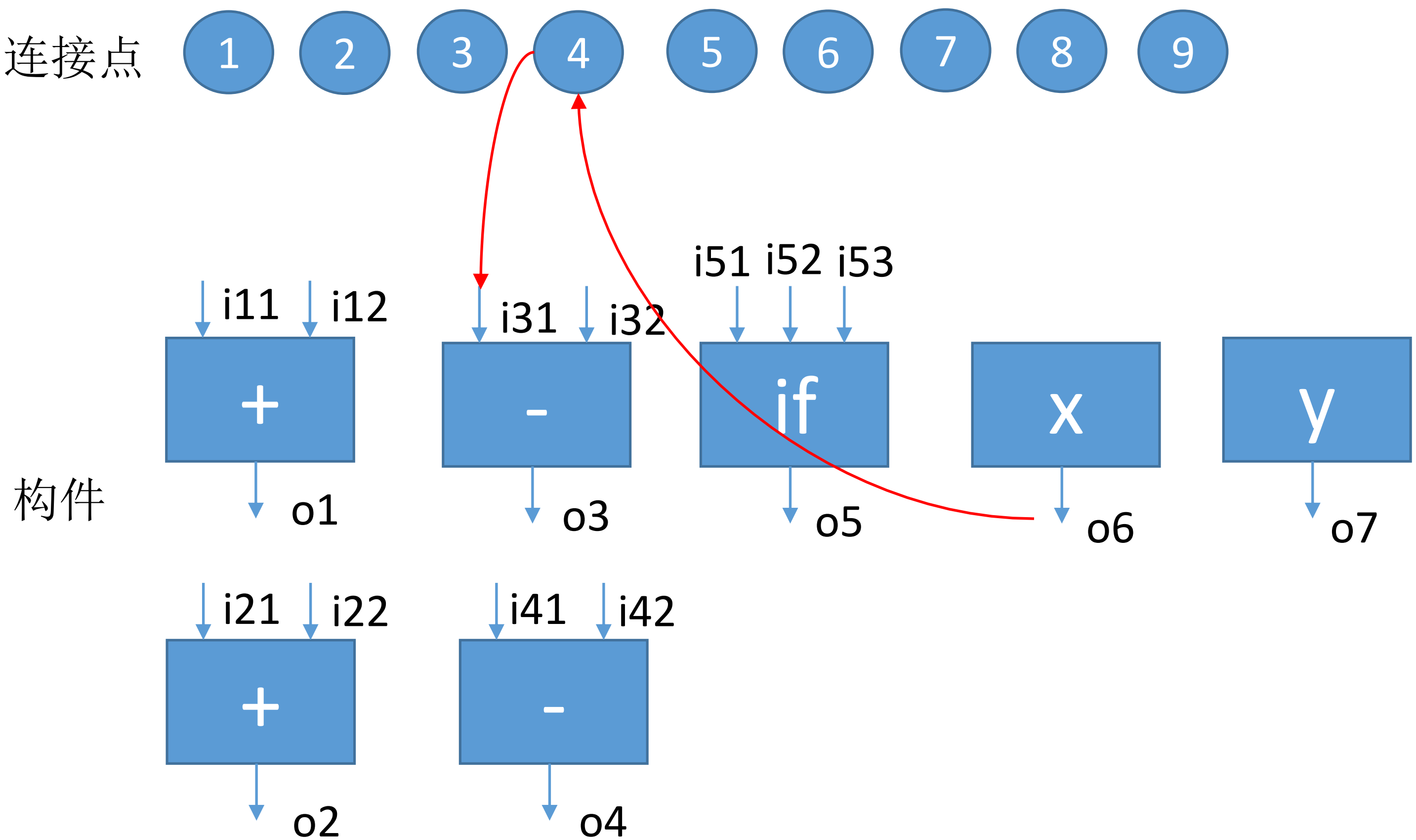
- 将程序合成问题整体转换成约束求解问题，由SMT求解器求解

Sumit Gulwani
微软研究院研究员
14年获SIGPLAN Robin Milner青年研究者奖

基于构件的程序合成
Component-Based Program Synthesis

添加标签变量：

- $l_{i11}, l_{i22}, \ldots$
- $l_{o1}, l_{o2}, \ldots$
- 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：
  - $o1 = i11 + i12$
- 对所有的输入输出标签对产生连接约束：
  - $l_{o1} = l_{i11} \rightarrow o_1 = i_{11}$
- 对所有的输出标签产生编号范围约束
  - $l_{o1} \geq 1 \wedge l_{o1} \leq 9$
- 对所有的o_i对产生唯一性约束
  - $l_{o1} \neq l_{o2}$
- 对统一构件的输入和输出产生防环约束
  - $l_{i11} < l_{o1}$

能否去掉连接点和输出标签l_{ox}...，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:
  - $x = 1 \land y = 2 \rightarrow o = 2$
  - $x = 3 \land y = 2 \rightarrow o = 3$
- 通过和CEGIS结合可以求解任意规约

基于抽象精化的合成

例子

- $n \to x \mid n + t \mid n \times t$
- $t \to 2 \mid 3$
- 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
- $[9]n_1 \to \boxed{[1]n + [8]t \mid [2]n + [7]t \mid \cdots}$
  $\mid [1]n \times [9]t \mid [3]n \times [3]t \mid [9]n \times [1]t$

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
  - 槑，即$x \in [-\infty, +\infty]$
  - $\ldots -7 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 8, 9 \leq \mathrm{x} \leq 18, \ldots$
  - $\ldots -3 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 4, 5 \leq \mathrm{x} \leq 8, \ldots$
  - $\ldots -1 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 2, 3 \leq \mathrm{x} \leq 4, \ldots$
  - $\ldots \mathrm{x} = -1, \mathrm{x} = 0, \mathrm{x} = 1, \ldots$
- 要求:
  - 包括槑，且γ(槑) = 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
- 一开始只包含槑，在精化过程中逐步增加

1.1 抽象域上的计算

- 抽象域包括 槑
- 自底向上构造VSA，得
  - [槑]n → [槑]x | [槑]n + [槑]t | [槑]n × [槑]t
  - [槑]t → [槑]2 | [槑]3
- 输入为x=槑，输出为ret=槑
- 随机从VSA中采样程序，得到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]

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

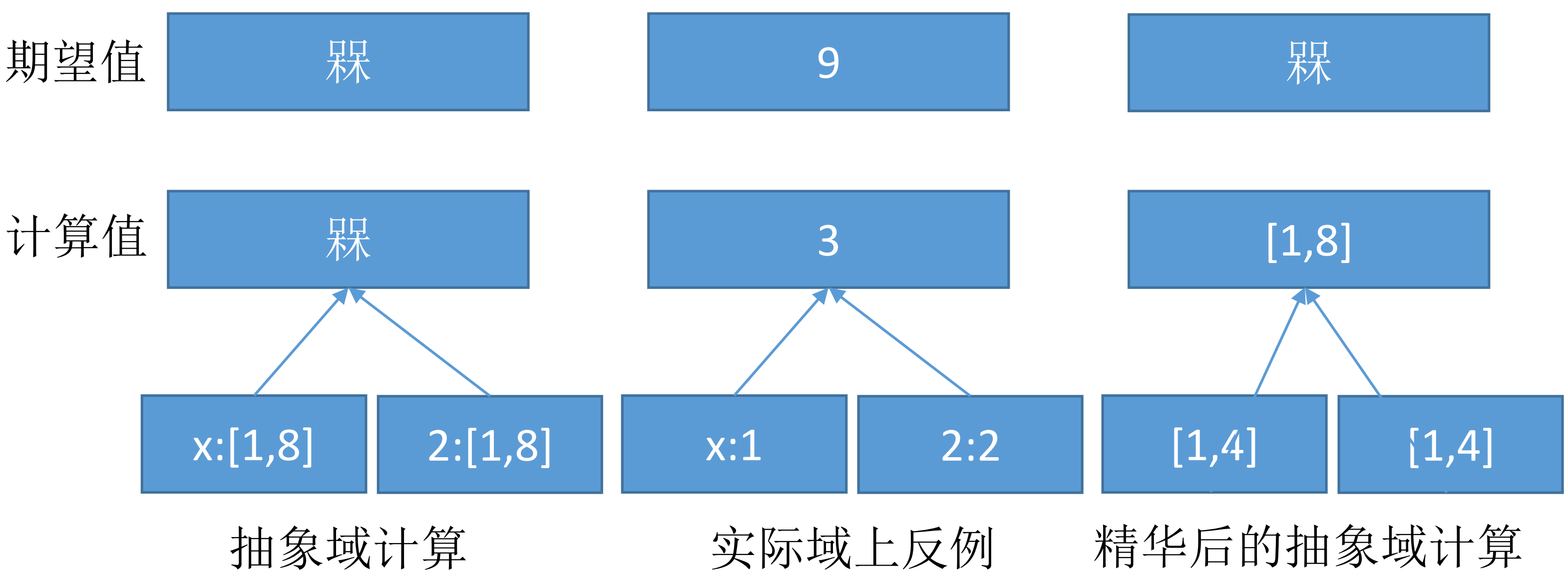
精化后抽象域的性质：
抽象域的运算结果一定包括反例输入在具体域上的运算结果
抽象域的运算结果一定不包括反例的期望输出

2.1 抽象域上的计算

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

计算过程的性质

- 给定反例e和精化后的抽象域虛，则
- 虛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
- 虛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.