软件分析

程序合成：约束求解和空间表示

熊英飞
北京大学

合一化程序合成STUN
Synthesis through Unification

- 针对If表达式的特殊合成方法
  - STUN系列求解器Eusolver, Euphony, PolyGen在SyGuS比赛的CLIA Track中表现最优
- 假设合成的程序具有如下形式
  - **if** boolExpr_1 **then** expr_1
    else if boolExpr_2 **then** expr_2
    …
    else expr_n
  - Expr_i是符合上面形式的表达式或不带if的原子表达式

Rajeev Alur
Upenn教授
STUN发明人，领导Eusolver和Euphony开发

基本流程(1/3)

- 首先枚举一组boolExpr和expr
  - 对于expr，记录所覆盖的example
  - 基于boolExpr，记录将example分成的两个组
  - 从小到大枚举，按概率枚举或者用专门的算法产生
- 假设枚举到x, y, 1三个表达式和x=3, x<y两个条件

x	y	ret	covering expr	boolExpr	
				x=3	x<y
1	2	2	y	false	true
3	3	3	x,y	true	false
5	2	5	x	false	false

基本流程(2/3)

- 选择覆盖所有例子的expr集合
  - 选择标准：限制expr大小的最大值、控制总大小等
- 在这里选择x和y

x	y	ret	covering expr	boolExpr	
				x=3	x<y
1	2	2	y	false	true
3	3	3	x,y	true	false
5	2	5	x	false	false

基本流程(3/3)

- 用决策树算法选择boolExpr构造最终程序
  - 选择boolExpr对example分类，让每个分类都能被一个所选expr覆盖
  - 选择标准：减少信息熵，或者控制总大小
- 选择x<y可以把原来的example分成两类，第一类包含第一个example，被x覆盖，第二类包含后两个example，被y覆盖
- 得到if(x<y) then x else y

x	y	ret	covering expr	boolExpr	
				x=3	x<y
1	2	2	y	false	true
3	3	3	x,y	true	false
5	2	5	x	false	false

奥卡姆程序合成

吉如一
北京大学博士生
奥卡姆程序合成
和PolyGen提出人

- 如何使用较少样例就得到正确程序?
  - 奥卡姆剃刀原则：如非必要，勿增实体
  - 映射到程序合成：合成尽量小的程序
- 严格要求最小程序导致合成算法效率低下
- 奥卡姆程序合成：高概率合成大小不超过最小程序多项式倍的程序
- 奥卡姆程序合成的用处
  - 提升CEGIS框架下的合成速度
  - 对于难以验证的规约，提供较高的概率错误保障

Eusolver/Euphony 不是奥卡姆程序合成

- Eusolver/Euphony控制了expr和boolExpr的大小，但没有控制他们的数量
- Eusolver/Euphony从小到大枚举expr
  - 可能导致用大量的小expr来覆盖一组样例
- 传统决策树算法是为模糊分类设计，用在程序合成上可能生成大量无用boolExpr
  - 假设三个表达式a, b, c可以覆盖所有样例且没有重叠
  - 某个boolExpr可以把样例分成如下两类
    - 类1：98个样例可以被a覆盖，1个样例被b覆盖，1个样例被c覆盖
    - 类2：98个样例被b覆盖，1个样例被a覆盖，1个样例被c覆盖
  - 该条件有很好的信息熵增益
  - 但对生成程序几乎没有帮助，因为每类还需要继续区分三个表达式

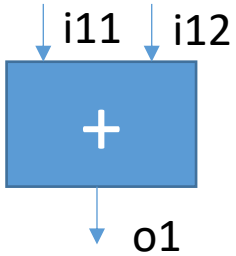
PolyGen

- 通过迭代加深算法控制expr的个数
  - 从一个较小的expr个数上限开始，如果合成失败，则给上限增加一个常量
- 通过生成决策列表而不是决策树来控制boolExpr的个数
  - **if** boolExpr_1 **then** expr_1
    else if boolExpr_2 **then** expr_2
    …
    else expr_n
- 在SyGuS的CLIA Track上， PolyGen的速度达到Eusolver和Euphony的6-15倍
- 详见PolyGen论文：
  - Ruyi Ji, Jingtao Xia, Yingfei Xiong, Zhenjiang Hu. Generalizable Synthesis Through Unification. OOPSLA'21: Object Oriented Programming Languages, Systems and Applications, October 2021.

约束求解法

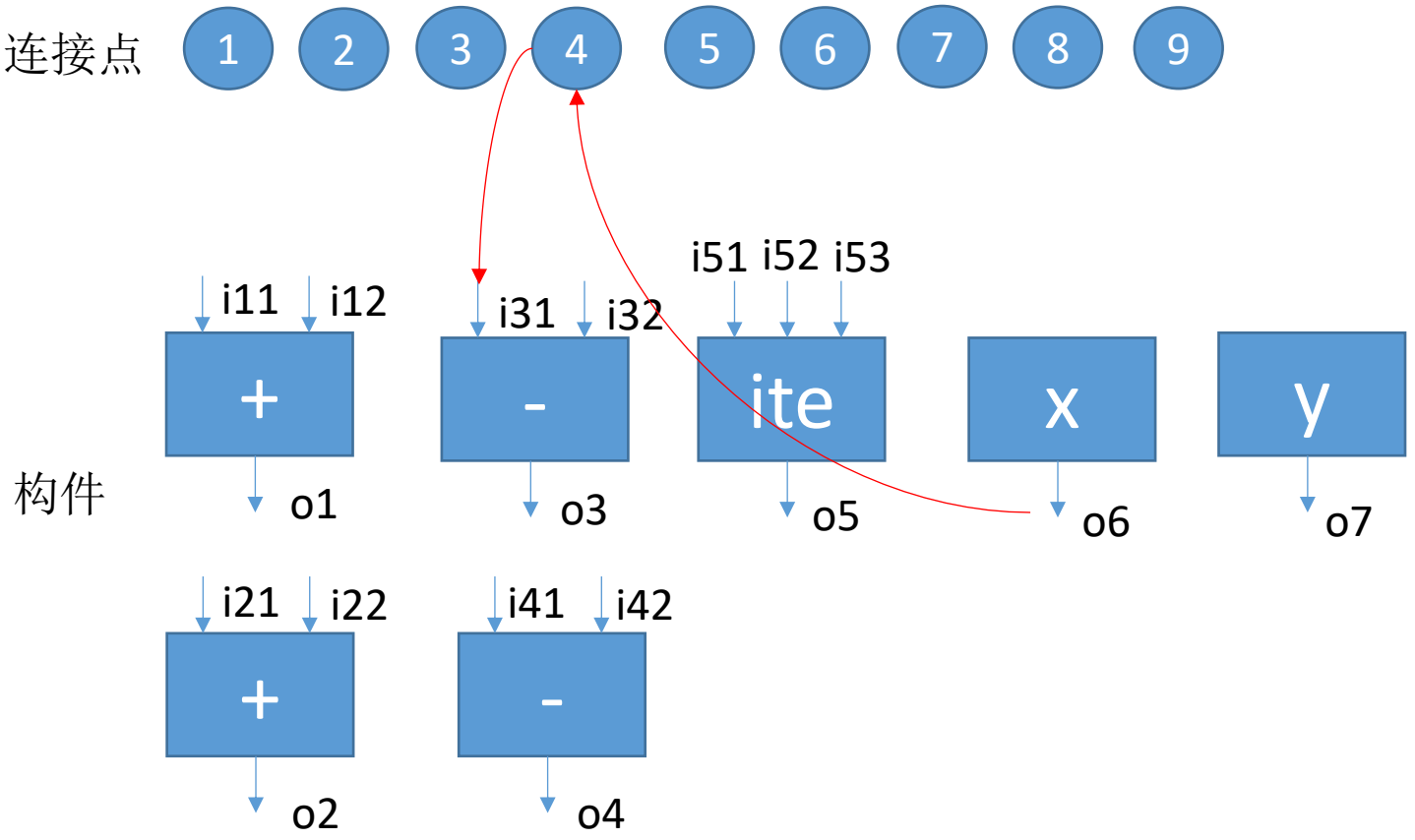
约束求解法

- 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
- 代表工作：基于构件的程序合成 Component-Based Program Synthesis
  - 获得ICSE十年最有影响力论文奖
- 将产生式看做构件，并且数量有限
  - 1号产生式：$Expr_{o1} \rightarrow Expr_{i11} + Expr_{i12}$



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



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：
  - $\forall x, y: o \geq x \wedge o \geq y \wedge (o = x \vee o = y)$
- 对所有构件和所有测试产生语义约束：
  - $v_{o1}^{t} = v_{i11}^{t} + v_{i12}^{t}$
- 对所有的输入输出标签对和所有测试产生连接约束：
  - $l_{o1} = l_{i21} \rightarrow v_{o1}^{t} = v_{i21}^{t}$
  - $l_{o1} = l_{i21} \rightarrow N_{o1} = N_{i21}$
- 对所有的输入输出标签产生编号范围约束:
  - $l_{o1} \geq 1 \wedge l_{o1} \leq 9$（假设总共9个构件）
  - $l_{i11} \leq 9$
- 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
  - $l_{i11} < l_{o1}$

额外变量：

- v_{o1}^{t}：测试t中o1位置的值
- N_{o1}：o1位置的非终结符

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：
  - $x = 1 \wedge y = 2 \rightarrow o = 2$
  - $x = 3 \wedge y = 2 \rightarrow o = 3$
- 通过和CEGIS结合可以求解任意规约

空间表示法

Sumit Gulwani
微软研究院研究员
14年获SIGPLAN Robin Milner青年研究者奖
提出自顶向下的VSA程序合成方法

王新宇
密西根大学
助理教授
提出自底向上的FTA程序合成方法

例子：化简的max问题

- 语法：

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

- 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

复习：可观察等价

- 可观察等价observational equivalence
  - 运行所有测试检测 $f = f'$
  - 认为等价的程序有可能不等价
    - 但在CEGIS框架中没有问题，因为我们目标是找到一个满足所有样例的程序而非满足完整规约的程序
  - 用于自底向上，是目前效果最好的等价性削减策略之一
- 为什么效果好?
  - 大量的程序在样例上都返回同样的结果，带来非常大的搜索空间节省
  - 如布尔表达式，单样例的时候最多有两个不等价程序

问题1：多样例的影响

- 样例越多，返回值的空间就越大
  - n个样例的布尔表达式有2^n个可能返回值
- 可观察等价的效果就越差

问题2：和CEGIS交互

- CEGIS每轮会新增一个样例
- 标准可观察等价剪枝无法利用之前的计算，会导致对之前的样例重复计算

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
- 对于集合依次求交得到最终程序的集合

- 效果
  - 单个样例时：可观察等价剪枝在单个样例时能发挥作用
  - 多个样例时：单个样例已经排除了大量程序，求交只用考虑部分程序
  - 样例逐步增加时：之前的求交结果保留，不会重复计算

基本思路

- 可观察等价思路：在当前样例中返回相同值且从同一个非终结符展开的程序是等价的
  - 用非终结符加上返回值约束来表示等价程序集合
    - 如：[2]Expr
  - 该表示可以看做一个新的非终结符，称为带约束非终结符
- 构建带约束上下文无关文法来表示程序空间，如：
  - [2]Expr → [1]Expr+[1]Expr
  - 对应的，称Expr → Expr+Expr为原始文法规则
- 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

如何从样例得到文法

- 在可观察等价剪枝的过程中构建文法
- 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
- 删除从起始符号和期望输出不可达的产生式

非终结符集合		产生式集合
[2]x	Expr→x	[2]Expr→[2]x
[1]y	Expr→ y	[1]Expr→[1]y
[2]Expr		
[1]Expr	Expr→Expr+Expr	[3]Expr→[2]Expr+[1]Expr
[3]Expr		

从样例得到文法例子1

- 语法:
  - S-> S + S | x | y | z
- 例子1:
  - ret="acc"
  - x="a"
  - y="cc"
  - z="c"
- 剪枝:
  - 生成的串不是ret的子串则剪掉

生成文法:

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

从样例得到文法例子2

- 语法：
  - S-> S + S | x | y | z
- 例子2：
  - ret=“aac”
  - x=“a”
  - y=“ac”
  - z=“c”
- 剪枝：
  - 生成的串不是ret的子串则剪掉

生成文法：

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

文法求交

- 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

- 本质原因：同一个字符串被解析成不同的AST树
- 解决思路：对语法树集合求交而不是对语言求交
  - 如果在原始文法上两颗语法树相同，则相同

文法求交1

- 给定原始产生式
  - $N_0 \to P(N_1, N_2, \dots)$
- 如果两个文法中分别存在两个带约束产生式
  - $[c_0]N \to P([c_1]N_1, [c_2]N_2, \dots)$
  - $[c_0']N \to P([c_1']N_1, [c_2']N_2, \dots)$
- 求交得到
  - $[c_0, c_0']N \to P([c_1, c_1']N_1, [c_2, c_2']N_2, \dots)$
- 对两个文法中所有这样的产生式求交，然后删掉无法从起始符号到达的产生式即可。
- 从效率考虑，可先从起始符号出发，只对从起始符号可达的产生式求交。

文法求交例子

[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~~

自顶向下的空间表示法合成

- 能否自顶向下构造文法?
- 反向语义:
  - 给定输出，什么样的输入能产生该输出
- 给定
  - [2]Expr
- 根据反向语义可以得到
  - [2]x, [2]y, [0]Expr+[2]Expr, [1]Expr+[1]Expr, [2]Expr+[0]Expr, ite([true]BoolExpr, [2]Expr, [*]Expr), ite([false]BoolExpr, [*]Expr, [2]Expr)
  - [*]表示任意值

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。

自顶向下构造文法

- 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 | …

自顶向下构造文法

- 根据witness function的实现，有可能出现非终结符无法展开的情况
- 文法生成后，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]Expr | [1]Expr+[2]Expr~~
- ~~[2]Expr→x|y~~
- ~~[1]Expr→ε~~

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

自底向上vs自顶向下

- 自顶向下最早出现在Excel的FlashFill系统
  - 采用VSA（Version Space Algebra上下文无关文法对求交封闭的子集）来表示空间
- 自底向上最早采用FTA（Finite Tree Automata有限树自动机）来表示空间
  - FTA本质上和VSA等价
- 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
- 实践中也可以把两种方法结合起来
  - 先用自底向上枚举一定数量的表达式，然后采用自顶向下的方法合成
  - Lee, Woosuk. "Combining the top-down propagation and bottom-up enumeration for inductive program synthesis." POPL (2021): 1-28.

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

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

- 之前见到的文法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

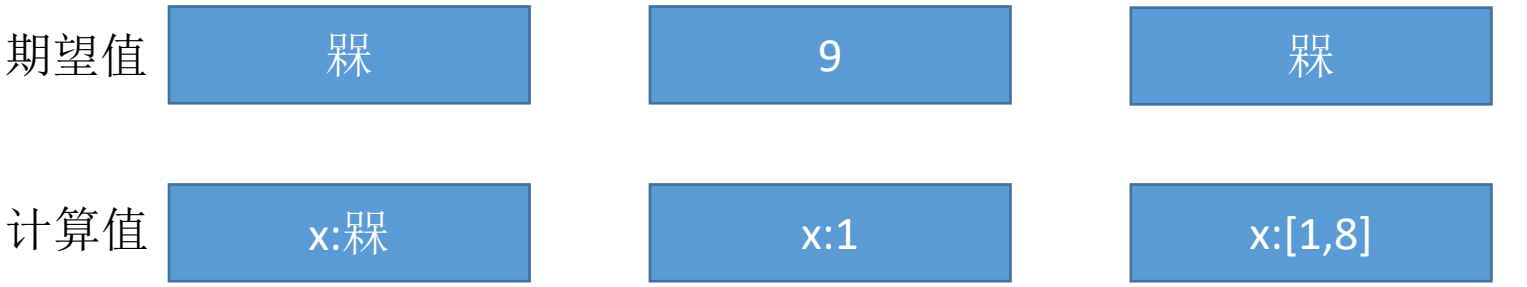
- 元抽象域由一组抽象值的集合构成，如
  - 粿，即$x \in [-\infty, +\infty]$
  - $\ldots -7 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 8, 9 \leq \mathrm{x} \leq 18, \ldots$
  - $\ldots -3 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 4, 5 \leq \mathrm{x} \leq 8, \ldots$
  - $\ldots -1 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 2, 3 \leq \mathrm{x} \leq 4, \ldots$
  - $\ldots \mathrm{x} = -1, \mathrm{x} = 0, \mathrm{x} = 1, \ldots$
- 要求：
  - 包括粿，且$\gamma(\text{粿}) =$ 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
- 一开始只包含粿，在精化过程中逐步增加

1.1 抽象域上的计算

- 抽象域包括 踩
- 自底向上构造文法，得
  - [踩]$n \rightarrow$ [踩]x | [踩]$n +$ [踩]t | [踩]$n \times$ [踩]t
  - [踩]$t \rightarrow$ [踩]2 | [踩]3
- 输入为x=踩，输出为ret=踩
- 随机从文法中采样程序，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

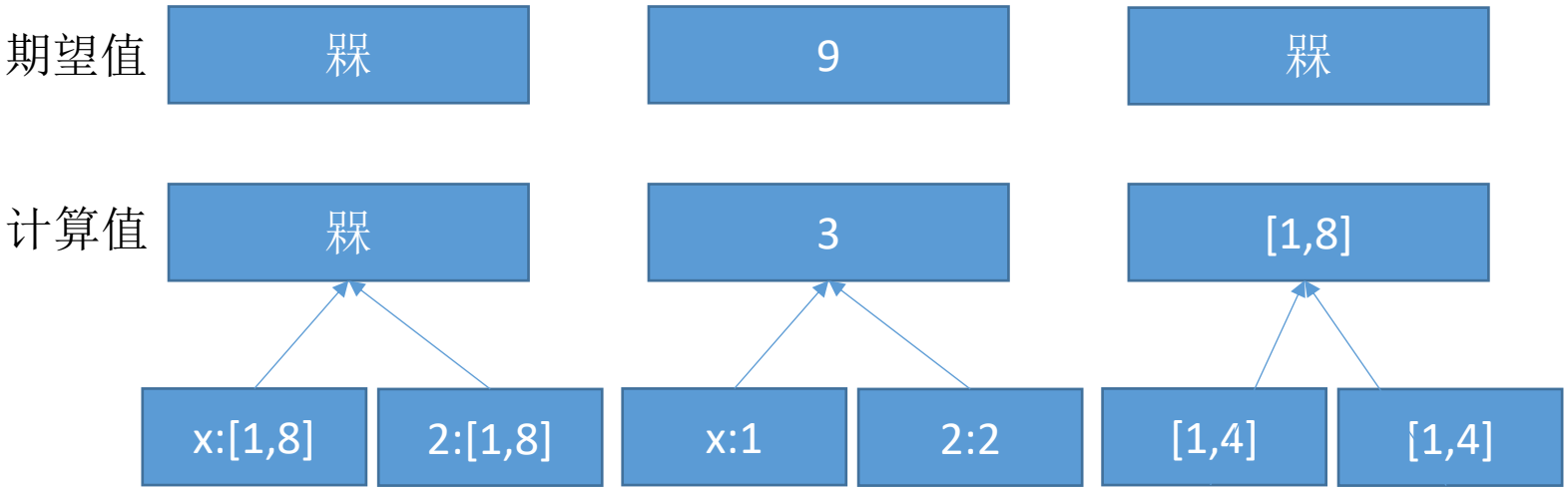
精化后抽象域的性质：
抽象域的运算结果一定包括反例输入在具体域上的运算结果
抽象域的运算结果一定不包括反例的期望输出

2.1 抽象域上的计算

- 抽象域包括 {踩, [1, 8]}
- 自底向上构造文法，得
  - [踩]$n \rightarrow$ [踩]n + [1,8]t | [踩]$n \times$ [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=踩
- 随机从文法中采样程序，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，根节点处理方式之前相同
  - 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 {踩, [1, 8],[1,4]}
- 自底向上构造文法，得
  - $[踩]n \rightarrow [踩]n + [1,4]t \mid [踩]n \times [1,4]t \mid [1,8]n + [1,4]t \mid [1,8]n \times [1,4]t \mid \cdots$
  - $[1,8]\ n \rightarrow [1,4]n + [1,4]t \mid \cdots$
  - $[1,4]\ n \rightarrow x$
  - $[1,4]\ t \rightarrow 2 \mid 3$
- 输入为x=[1,4]，输出为ret=踩
- 随机从文法中采样程序，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
- 即使最后的文法也比完整的文法小很多，实现加速

参考文献

- Rajeev Alur, Pavol Cerný, Arjun Radhakrishna: Synthesis Through Unification. CAV (2) 2015: 163-179
- Ruyi Ji, Jingtao Xia, Yingfei Xiong, Zhenjiang Hu. Generalizable Synthesis Through Unification. OOPSLA'21: Object Oriented Programming Languages, Systems and Applications, October 2021.
- 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.