软件分析

程序综合：约束求解和空间表示

熊英飞

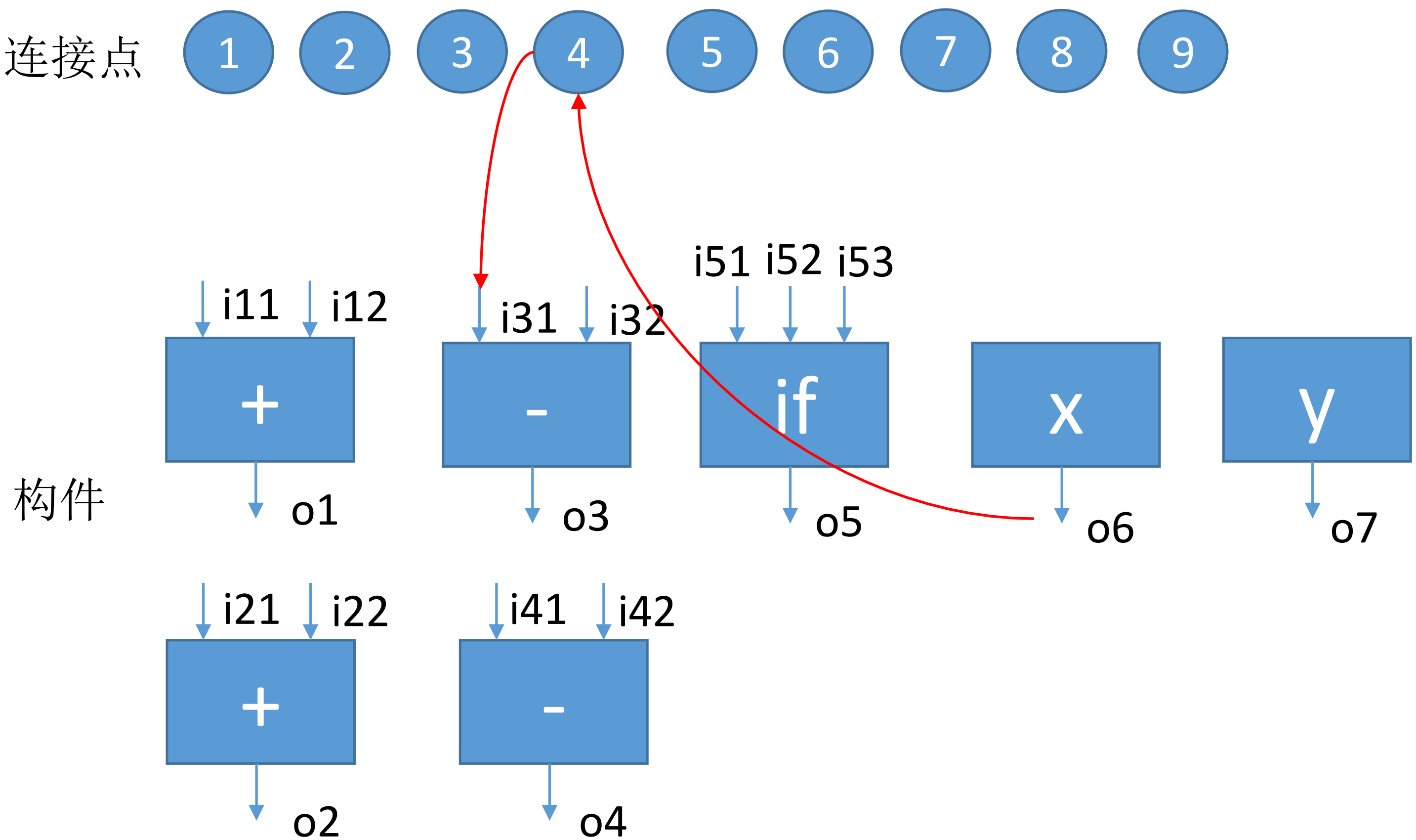
北京大学

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：

- $l_{i11}, l_{i22}, \ldots$
- $l_{o1}, l_{o2}, \ldots$
- 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：
  - $x = 1 \wedge y = 2 \rightarrow o = 2$
  - $x = 3 \wedge y = 2 \rightarrow o = 3$
- 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:

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

- 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

自顶向下遍历

- 按语法依次展开
  - Expr
  - x, y, Expr+Expr, if(BoolExpr, Expr, Expr)
  - y, Expr+Expr, if(BoolExpr, Expr, Expr)
  - Expr+Expr, if(BoolExpr, Expr, Expr)
  - x+Expr, y+Expr, Expr+Expr+Expr, if(BoolExpr, Expr, Expr)+Expr, if(BoolExpr, Expr, 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函数细化为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]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
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:

$$
\begin{aligned}
S &\to AC \\
A &\to aAb \mid ab \\
C &\to cC \mid c
\end{aligned}
\qquad
\begin{aligned}
S' &\to A'C' \\
A' &\to aA' \mid a \\
C' &\to bC'c \mid bc
\end{aligned}
$$

$S \cap S'$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
  - $N \rightarrow p_1 \mid p_2 \mid \cdots \mid p_n$
  - $N \rightarrow N_1 \mid N_2 \mid \cdots \mid N_n$
  - $N \rightarrow f(N_1, N_2, \dots, N_n)$
  - 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

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

- Expr ::= V | Add |
- 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

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

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]Expr | [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

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

VSA求交

- 如果$N \to f(N_1 \mid \dots \mid N_k)$且$N' \to p'_1 \mid p'_2 \mid \cdots$且存在$p'_{j1}, \dots, p'_{jm}$使得$N$可以生成$p'_{j1}, \dots, p'_{jm}$中的每一个而不能生成别的
  - $intersect(N, N') = \{N \cap N' \to p'_{j1}, \dots, N \cap N' \to p'_{jm}\}$
  - 否则$intersect(N, N') = \{N \cap N' \to \perp\}$
- 如果$N \to p_1 \mid p_2 \mid \cdots$且$N' \to p'_1 \mid p'_2 \mid \cdots$且$p''_{j1}, \dots, p''_{jm}$是两者的公共部分
  - $intersect(N, N') = \{N \cap N' \to p''_{j1}, \dots, N \cap N' \to p''_{jm}\}$
- 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

多个样例的情况

- 每个样例产生VSA，然后求交
- 在一个VSA上用另外一个样例做过滤
  - 第二个样例上witness函数不能展开的选项就去掉
  - 在CEGIS的时候可以加快速度
- 两个样例同时生成
  - 两个样例同时产生的选项才保留
  - 在规约是样例的时候可以加快速度

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		产生式集合
[2]x	Expr→x	[2]Expr→[2]x
[1]y	Expr→ y	[1]Expr→[1]y
[2]Expr	Expr→Expr+Expr	[3]Expr→[2]Expr+[1]Expr
[1]Expr		
[3]Expr		

自底向上vs自顶向下

- 两种方法有不同的实用范围
  - 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
- $[9]n_1 \rightarrow \boxed{[1]n + [8]t \mid [2]n + [7]t \mid \cdots}$
  $\mid [1]n \times [9]t \mid [3]n \times [3]t \mid [9]n \times [1]t$

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

基于区间的元抽象域

- 槑，即$x \in [-\infty, +\infty]$
- $\dots -7 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 8, 9 \leq \mathrm{x} \leq 18, \dots$
- $\dots -3 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 4, 5 \leq \mathrm{x} \leq 8, \dots$
- $\dots -1 \leq \mathrm{x} \leq 0, 1 \leq \mathrm{x} \leq 2, 3 \leq \mathrm{x} \leq 4, \dots$
- $\dots \mathrm{x} = -1, \mathrm{x} = 0, \mathrm{x} = 1, \dots$

- 元抽象域由以上抽象值构成
  - 假设：针对任意包含槑的抽象值的集合均能构造抽象运算
- 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
- 构造VSA，得
  - [槑]$n \rightarrow x$ | [槑]$n +$ [槑]t | [槑]$n \times$ [槑]t
  - [槑]$t \rightarrow 2$ | 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]

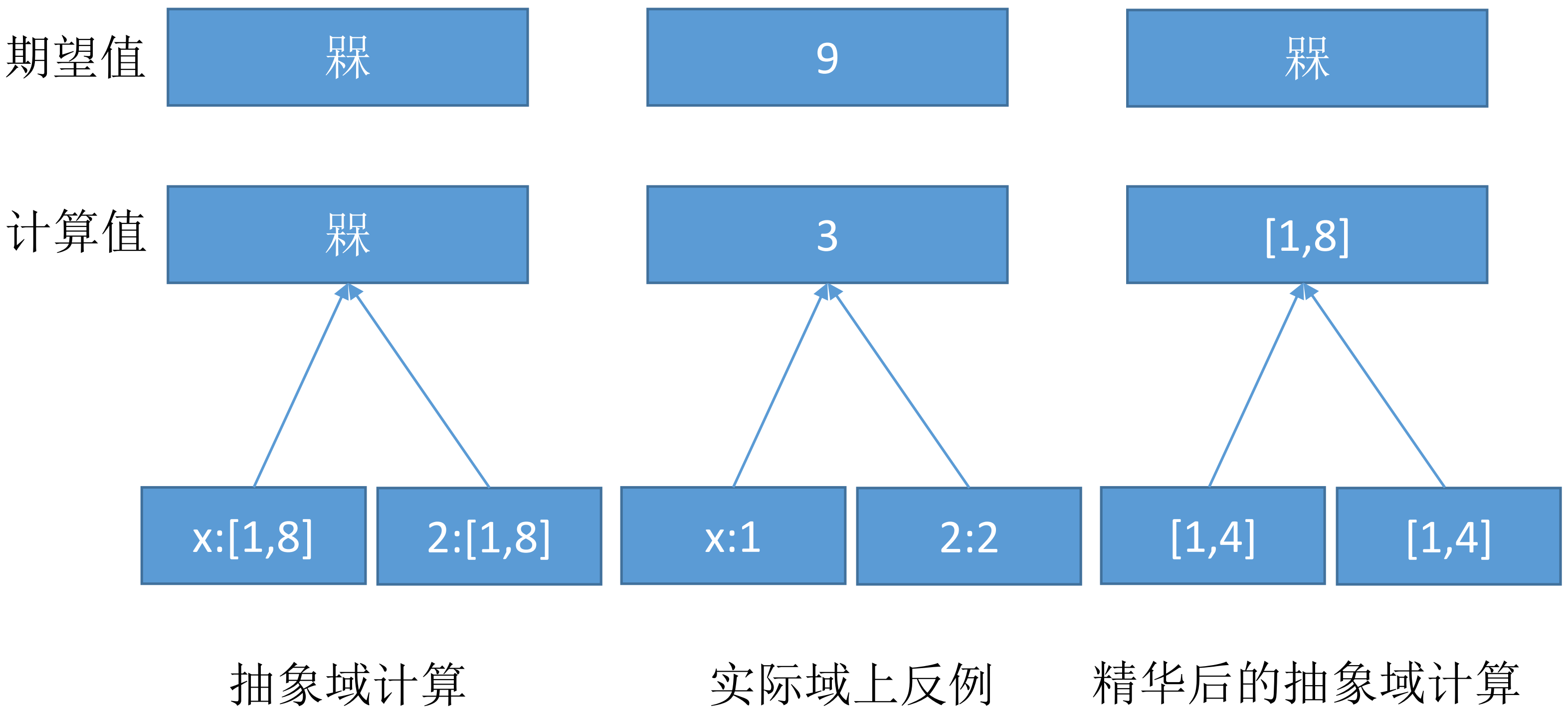
期望值	槑	9	槑
计算值	x:槑	x:1	x:[1,8]
	抽象域计算	实际域上反例	精华后的抽象域计算

2.1 抽象域上的计算

- 抽象域包括 {槑, [1, 8]}
- 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] $n \rightarrow x$
  - [1,8] $t \rightarrow 2$ | 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，使得该抽象值列表包括计算值，且抽象域上计算结果为当前结点
- 添加抽象值[1, 4]



3.1 抽象域上的计算

- 抽象域包括 {槑, [1, 8],[1,4]}
- 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

参考文献

- 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.