



归纳程序合成

熊英飞
北京大学

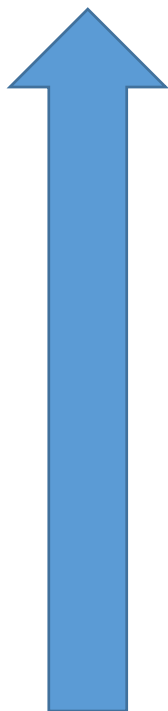


外祖母可以编程吗？

- 程序设计语言的发展历史就是提高抽象级别



抽象级别



外祖母编程语言？

Haskell (1990), Prolog (1972)

Java

C

Assembly



为什么外祖母还不能编程？

- 程序设计语言默认保证很多属性
 - 类型正确的程序一定能通过编译
 - 通过编译的程序有清晰定义的语义
- 很难再进一步提升抽象级别





程序合成——外祖母的希望

- 从规约中自动生成程序
 - 规约可能是模糊的
 - 生成是不保证成功的



“One of the most central problems in the theory of programming.”

----Amir Pnueli
图灵奖获得者

“（软件自动化）提升软件生产率的根本途径”

----徐家福先生
中国软件先驱



程序合成的历史

1957

- 程序合成的开端
- Alonzo Church: 电路合成问题

2000前

- 演绎合成

2000后

- 归纳合成

典型应用——Data Wrangling

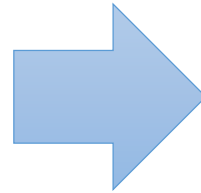


| | A | B |
|----|--------------------------------------------|---------------------|
| 1 | Email | Column 2 |
| 2 | Nancy.FreeHafer@fourthcoffee.com | nancy freehafer |
| 3 | Andrew.Cencici@northwindtraders.com | andrew cencici |
| 4 | Jan.Kotas@litwareinc.com | jan kotas |
| 5 | Mariya.Sergienko@gradicdesigninstitute.com | mariya sergienko |
| 6 | Steven.Thorpe@northwindtraders.com | steven thorpe |
| 7 | Michael.Neipper@northwindtraders.com | michael neipper |
| 8 | Robert.Zare@northwindtraders.com | robert zare |
| 9 | Laura.Giussani@adventure-works.com | laura giussani |
| 10 | Anne.HL@northwindtraders.com | anne hl |
| 11 | Alexander.David@contoso.com | alexander david |
| 12 | Kim.Shane@northwindtraders.com | kim shane |
| 13 | Manish.Chopra@northwindtraders.com | manish chopra |
| 14 | Gerwald.Oberleitner@northwindtraders.com | gerwald oberleitner |
| 15 | Amr.Zaki@northwindtraders.com | amr zaki |
| 16 | Yvonne.McKay@northwindtraders.com | yvonne mckay |
| 17 | Amanda.Pinto@northwindtraders.com | amanda pinto |

典型应用– Superoptimization

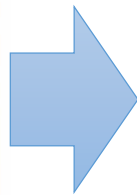


```
i=round(i);
```



```
a = 6755399441055744.0;  
i=(i+a)-a;
```

典型应用-自动编写重复程序



```
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
            CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
            battlecry=Battlecry(Destroy(),
                WeaponSelector(EnemyPlayer()))))

    def create_minion(self, player):
        return Minion(3, 2)
```




典型应用-缺陷修复

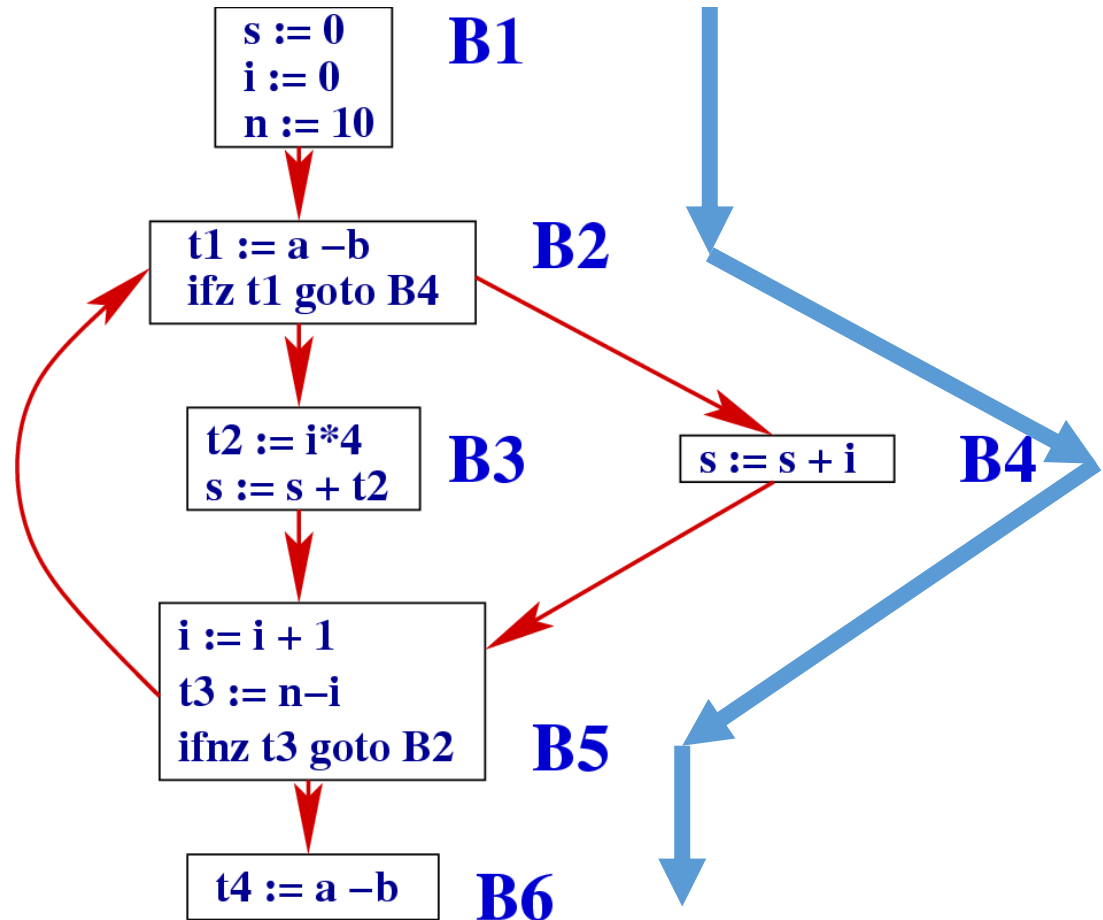
```
/** Compute the maximum of two values
 * @param a first value
 * @param b second value
 * @return b if a is lesser or equal to b, a otherwise
 */
public static int max(final int a, final int b) {
    return (a <= b) ? a : b;
}
```

合成出新的表达式来替换掉旧的



典型应用-生成测试

合成单元测试
来覆盖某路径





典型应用-加速程序分析

SMT Solver

```
Apply Tactic 1
If formula is long
  Apply Tactic 2
Else
  Apply Tactic 3
```

策略

针对一组问题合成最佳策略



程序合成定义

- 输入:
 - 一个程序空间 $Prog$, 通常用语法表示
 - 一条规约 $Spec$, 通常为逻辑表达式
- 输出:
 - 一个程序 $prog$, 满足
 - $prog \in Prog \wedge prog \vdash Spec$
- 局限性:
 - 当规约是模糊或者不完整的时候, 正确性就完全无保障了



例子：max问题

- 语法：

```
Expr ::= 0 | 1 | x | y
      | Expr + Expr
      | Expr - Expr
      | (ite BoolExpr Expr Expr)
BoolExpr ::= BoolExpr ^ BoolExpr
          | ¬BoolExpr
          | Expr ≤ Expr
```

- 规约：

$$\forall x, y : \mathbb{Z}, \quad \text{max}_2(x, y) \geq x \wedge \text{max}_2(x, y) \geq y \\ \wedge (\text{max}_2(x, y) = x \vee \text{max}_2(x, y) = y)$$

- 期望答案： $\text{ite}(x \leq y) y x$



SyGuS: 程序合成问题的标准化

- 输入：语法 G ，约束 C
- 输出：程序 P ， P 符合语法 G 并且满足 C

- 输入输出格式：Synth-Lib
 - <http://sygus.seas.upenn.edu/files/SyGuS-IF.pdf>



Sync-Lib: 定义逻辑

- 和SMT-Lib完全一致
- (set-logic LIA)
- 该逻辑定义了我们后续可以用的符号以及这些符号的语法/语义，程序的语法应该是该逻辑语法的子集。



Sync-Lib: 语法

```
(synth-fun max2 ((x Int) (y Int)) Int
  ((Start Int (x
    y
    0
    1
    (+ Start Start)
    (- Start Start)
    (ite StartBool Start Start)))
  (StartBool Bool ((and StartBool StartBool)
    (or StartBool StartBool)
    (not StartBool)
    (<= Start Start)
    (= Start Start)
    (>= Start Start)))))
```




约束

```
(declare-var x Int)
(declare-var y Int)
```

约束表示方式和SMTLib一致

```
(constraint (>= (max2 x y) x))
(constraint >= (max2 x y) y)
(constraint(or (= x (max2 x y))
               (= y (max2 x y))))
```

```
(check-synth)
```



期望输出

输出:

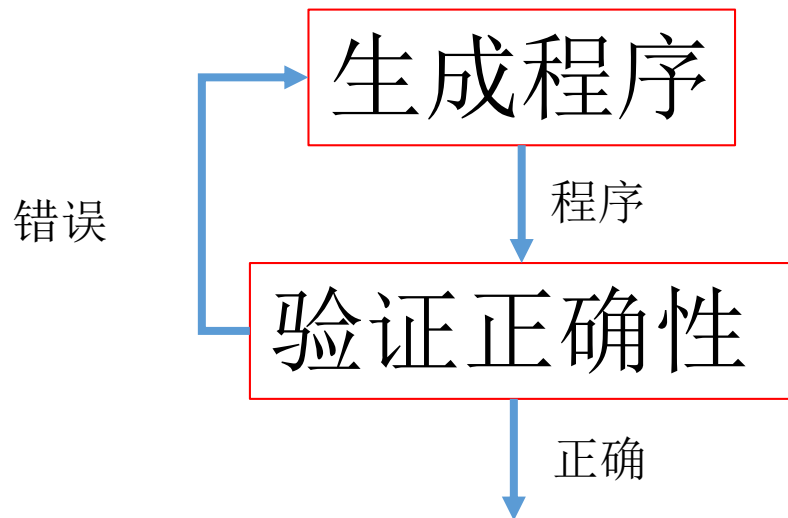
```
(define-fun max2 ((x Int) (y Int)) Int (ite (<= x y) y x))
```

输出必须:

- 满足语法要求
 - 即，语法和SMTLib/Logic不一致就合成不出正确的程序
- 满足约束要求
 - 一般要求可以通过SMT验证

归纳程序合成

——程序空间上搜索



Q1:如何产生下一个被搜索的程序?

Q2:如何验证程序的正确性?

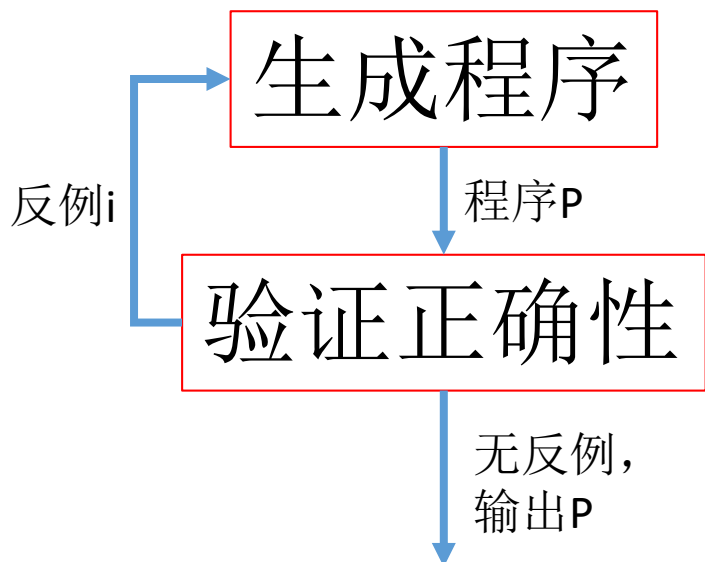


如何验证程序的正确性?

- 目前大多数程序合成技术都只处理表达式
 - 可直接转成约束让SMT求解
 - Synth-lib直接提供支持



CEGIS——基于反例的优化



- 采用约束求解验证程序的正确性较慢
- 执行测试较快
 - 大多数错误被一两个测试过滤掉
- 将约束求解器返回的反例作为测试输入保存
- 验证的时候首先采用测试验证



Armando Solar-Lezama
麻省理工大学教授

如何产生下一个被搜索的程序？



- 多种不同方法
 - 枚举法 —— 按照固定格式搜索
 - 约束求解法 —— 转成约束求解问题
 - 空间表示法 —— 一次考虑一组程序而非单个程序
 - 基于概率的方法 —— 基于概率模型查找最有可能的程序



枚举法



自顶向下遍历

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



自底向上遍历

- 从小到大组合表达式
 - size=1
 - x, y
 - size=2
 - size=3
 - $x+y, x-y$
 - size=4
 - size=5
 - $x+(x+y), x-(x+y), \dots$
 - size=6
 - $(\text{ite } x \leq y, x, y), \dots$



优化

- 等价性削减
 - 如果等价于一个之前的程序，则停止展开。
 - $\text{Expr}+x, \text{x+Expr}$
- 剪枝
 - 如果所有对应完整程序都不能满足约束，则停止展开
 - $\text{Ite BoolExpr } x \ x$




判断程序是否等价

- 通过SMT求解器可以判断
 - 判断 $f(x, y) \neq f'(x, y)$ 是否可以满足
 - 开销较大，不一定划算
- 通过测试判断
 - 运行所有测试检测 $f = f'$
 - 并不能保证结果的正确性
 - 对于不完整程序不能运行测试
- 通过预定义规则判断
 - 如 $S+x$ 和 $x+S$ 的等价性




剪枝基本方法：约束求解

- 从部分程序中生成约束
 - 针对每个组件预定义约束
 - 该约束可以不充分但必须必要
 - 根据语法树将约束连接在一起

Ite BoolExpr x x  (declare-fun boolExpr () Int)
(declare-fun max2 ((x Int) (y Int)) Int
(ite boolExpr x x))

- 从测试中生成约束

max2(1,2)=2  (assert (= (max2 1 2) 2))
(check-sat)



剪枝的优化

- 剪枝起作用的条件
 - 剪枝的分析时间 $<$ 被去掉程序的分析时间
 - 约束求解的开销通常较大
- 如何快速分析出程序不满足约束?
- 预分析
 - 在语法上离线做静态分析
 - 根据静态分析的结果快速在线剪枝



语法上的静态预分析

- 假设所有约束都是 $\text{Pred}(\text{Prop}(N))$ 的形式
 - N : 非终结符
 - Prop : 以 N 为根节点的子树所具有的属性值
 - Pred : 该属性值所应该满足的谓词
- 如:
 - 语义约束: Prop 为表达式取值
 - 类型约束: Prop 为表达式的可能类型
 - 大小约束: Prop 为表达式的大小
- 通过静态分析获得 Prop 的所有可能取值
 - 要求上近似
- 如果所有可能取值都不能满足 Pred , 则该部分程序可以减掉



语法上静态分析示例：语义

- 抽象域：由0, 1, 2, 3, >3, <0, true, false构成的集合
- 容易定义出抽象域上的计算
- 给定输入输出样例 $x=1, y=0, \max2(x,y)=1$
- 从语法规则产生方程
- $E \rightarrow E+E \mid 0 \mid 1 \mid x \mid \dots$
 - $V[E] = (V[E]+V[E]) \cup \{0\} \cup \{1\} \cup \{1\} \dots$
- 求解方程得到每一个非终结符可能的取值（在开始时做一次）
- 根据当前的部分程序产生计算式

ite BoolExpr x x  $V[E] = V[x] \cup V[x]$



语法上静态分析示例：类型

- 抽象域：由Int, String, Boolean构成的集合

- 从语法规则产生方程

- $E \rightarrow E + E \mid 0 \mid 1 \mid x \mid \dots$

- $T[E] = (T[E] + T[E]) \cup \{Int\} \cup \{Int\} \cup \{Int\} \dots$

- 其中

- $t_1 + t_2 = \begin{cases} \{Int\}, & Int \in t_1 \wedge Int \in t_2 \\ \emptyset, & \text{否则} \end{cases}$



语法上静态分析示例：大小

- 抽象域：整数
- 从语法规则产生方程
- $E \rightarrow E + E \mid 0 \mid 1 \mid x \mid \dots$
 - $S[E] = \min(2S[E], 1, 1, 1, \dots)$



冲突制导的在线学习

- 之前的剪枝方法主要依赖离线分析
- 在线搜索的过程中如果遇到不满足规约的程序，能否从中学习到更好的剪枝方法？
 - 现代SAT/SMT的主要技术思想之一



冯煜
UCSB助理教授



例子：序列操作合成

$N \rightarrow \emptyset \mid \dots \mid 1\emptyset \mid x_i \mid \text{last}(L) \mid \text{head}(L) \mid \text{sum}(L)$
 $\mid \text{maximum}(L) \mid \text{minimum}(L)$

$L \rightarrow \text{take}(L, N) \mid \text{filter}(L, T) \mid \text{sort}(L) \mid \text{reverse}(L) \mid x_i$

$T \rightarrow \text{geqz} \mid \text{leqz} \mid \text{eqz}$

输入： $x=[49, 62, 82, 54, 76]$

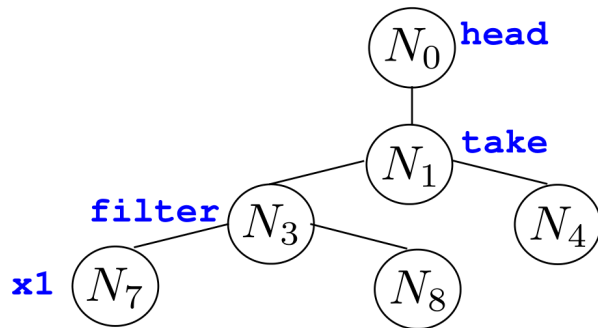
输出： $y=158$



用基本方法剪枝部分程序

| Component | Specification |
|-----------|-----------------------------------------------------------------------------------------|
| head | $x_1.size > 1 \wedge y.size = 1 \wedge y.max \leq x_1.max$ |
| take | $y.size < x_1.size \wedge y.max \leq x_1.max \wedge$ $x_2 > 0 \wedge x_1.size > x_2$ |
| filter | $y.size < x_1.size \wedge y.max \leq x_1.max$ |

head(take(filter(x1, T), N))



$$x_1 = [49, 62, 82, 54, 76] \wedge y = 158$$

$$\phi_{N_0} = \underline{y \leq v_1.max} \wedge v_1.size > 1 \wedge y.size = 1$$

$$\phi_{N_1} = \underline{v_1.max \leq v_3.max} \wedge v_1.size < v_3.size \wedge$$

$$v_4 > 0 \wedge v_3.size > v_4$$

$$\phi_{N_3} = v_3.size < v_7.size \wedge \underline{v_3.max \leq v_7.max}$$

$$\phi_{N_7} = \underline{x_1 = v_7}$$

加下划线的为极小矛盾集 ψ

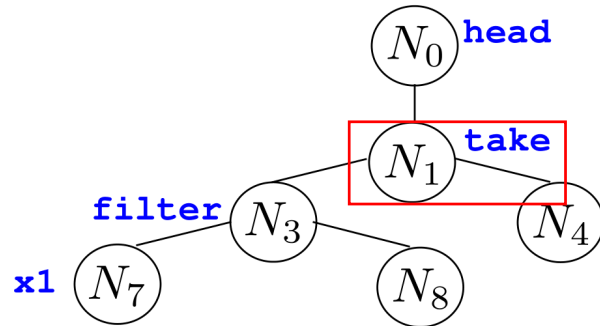


从冲突中学习

- 如果
 - 从新程序P导出的规约 \Rightarrow 该矛盾集 ψ
- 则
 - P也是无效程序
- 用约束求解器判断上述条件不一定比直接判断新程序是否满足规约快
 - 如何快速排除部分程序?



对当前冲突等价



$$\phi_{N_0} = \underline{y \leq v_1.max} \wedge v_1.size > 1 \wedge y.size = 1$$

$$\phi_{N_1} = \underline{v_1.max \leq v_3.max} \wedge v_1.size < v_3.size \wedge v_4 > 0 \wedge v_3.size > v_4$$

$$\phi_{N_3} = v_3.size < v_7.size \wedge \underline{v_3.max \leq v_7.max}$$

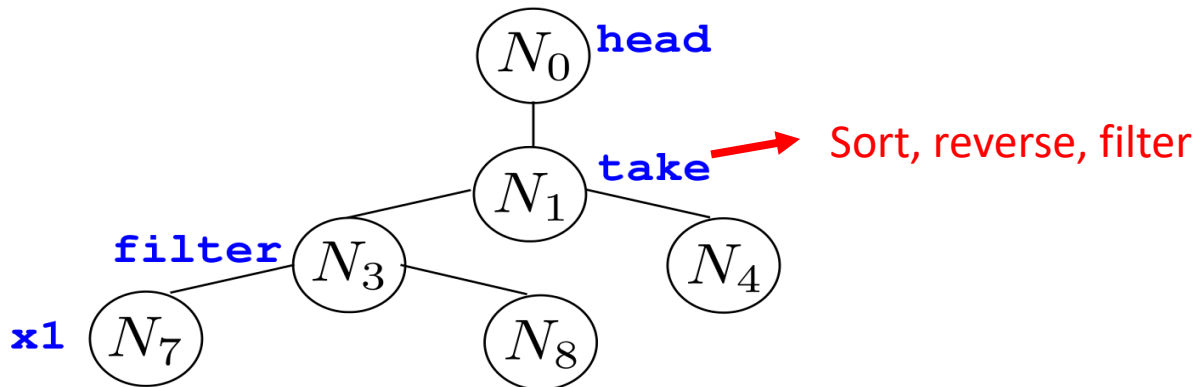
$$\phi_{N_7} = \underline{x_1 = v_7}$$

- N_1 位置的take因为 $y.max \leq x_1.max$ 而冲突
- 任意组件f, 如果
 - f的规约 $\Rightarrow y.max \leq x_1.max$
- 则
 - f在 N_1 位置和当前冲突等价
- 因为只涉及到组件的固定规约, 可以用SMT solver离线验证



排除程序

- 遍历组件可以发现，`sort`, `reverse`, `filter`都在 N_1 位置和当前冲突等价



- 所有将 N_1 替换这些组件的程序都无效
- 考虑其他位置的等价以及这些等价关系的组合，能排除较多等价程序。



空间表示法



例子：化简的max问题

- 语法：

$$\begin{array}{l} \text{Expr} ::= x \mid y \\ \quad \quad | \text{Expr} + \text{Expr} \\ \quad \quad | (\text{ite BoolExpr Expr Expr}) \\ \text{BoolExpr} ::= \text{BoolExpr} \wedge \text{BoolExpr} \\ \quad \quad | \neg \text{BoolExpr} \\ \quad \quad | \text{Expr} \leq \text{Expr} \end{array}$$

- 规约：

$$\forall x, y : \mathbb{Z}, \quad \text{max}_2(x, y) \geq x \wedge \text{max}_2(x, y) \geq y \\ \wedge (\text{max}_2(x, y) = x \vee \text{max}_2(x, y) = y)$$

- 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
 - $[2]y, [1]\text{Expr}+[1]\text{Expr}, \text{if}([\text{true}]\text{BoolExpr}, [2]\text{Expr}, [*]\text{Expr}), \text{if}([\text{false}]\text{BoolExpr}, [*]\text{Expr}, [2]\text{Expr})$
- Witness Function 需要由用户提供
- 但针对每个 DSL 只需要提供一次



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



基于空间表示的合成

- 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：
 - $[2]\text{Expr} \rightarrow [2]y \mid [1]\text{Expr}+[1]\text{Expr}$
 - 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
```

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

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid ab$$

$$C \rightarrow cC \mid c$$

- 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

- $[N \cap N']$ 表示把 N 和 N' 求交之后的终结符
- 如果 $N \rightarrow N_1 \mid N_2 \mid \dots$
 - $[N \cap N'] \rightarrow [N_1 \cap N'] \mid [N_2 \cap N'] \mid \dots$
- 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 $k \neq k'$
 - $[N \cap N'] \rightarrow \epsilon$
- 如果 $N \rightarrow f(N_1 \mid \dots \mid N_k)$ 且 $N' \rightarrow f(N'_1 \mid \dots \mid N'_k)$
 - $[N \cap N'] \rightarrow f([N_1 \cap N'_1], \dots, [N_k \cap N'_k])$



VSA求交

- 如果 $N \rightarrow p_1 \mid p_2 \mid \dots$ ，则将 N' 全部展开，和 $\{p_1, p_2, \dots\}$ 求交得到 $\{p'_{j_1}, p'_{j_2}, \dots\}$
 - $[N \cap N'] \rightarrow p'_{j_1} \mid p'_{j_2} \mid \dots$
- 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：
 - $S \rightarrow S + S \mid x \mid y \mid z$
- 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:

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

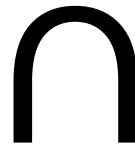
生成VSA:

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



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



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

$\text{Expr} \rightarrow x$
 $\text{Expr} \rightarrow y$
 $\text{Expr} \rightarrow \text{Expr} + \text{Expr}$

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



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
14年获SIGPLAN
Robin Milner青年
研究者奖

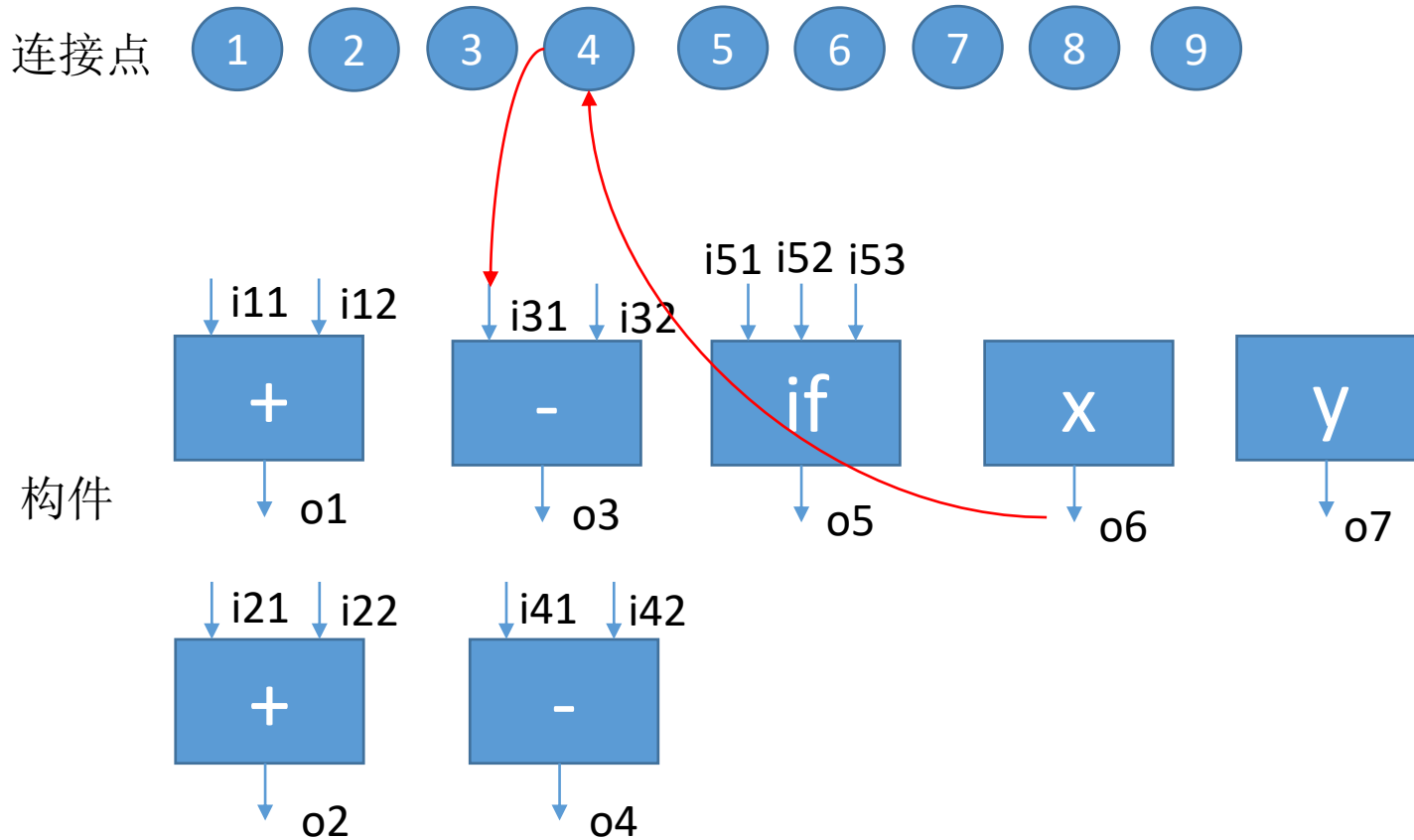
基于构件的程序合成

Component-Based Program Synthesis



添加标签变量:

- l_{i11}, l_{i22}, \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



基于概率的方法

很多应用需要概率最大的程序



典型应用-自动编写重复程序



```
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
            CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
            battlecry=Battlecry(Destroy(),
                WeaponSelector(EnemyPlayer()))))
    def create_minion(self, player):
        return Minion(3, 2)
```

典型应用-缺陷修复



```
/** Compute the maximum of two values
 * @param a first value
 * @param b second value
 * @return b if a is lesser or equal to b, a otherwise
 */
public static int max(final int a, final int b) {
    return (a <= b) ? a : b;
}
```

综合出新的表达式来替换掉旧的

10



程序估计 Program Estimation

- 输入:
 - 一个程序空间 $Prog$
 - 一条规约 $Spec$
 - 概率模型 P ，用于计算程序的概率
- 输出:
 - 一个程序 $prog$ ，满足
 - $prog = \operatorname{argmax}_{prog \in Prog \wedge prog \vdash spec} P(prog)$
- 如果 P 估计程序满足规约的概率，那么可以用来加速传统程序合成



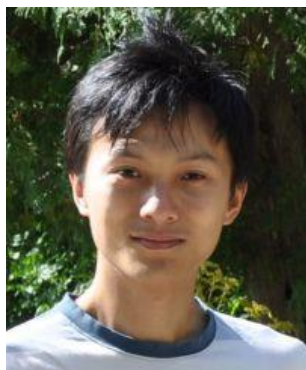
基本算法：穷举

- 用枚举的方法遍历空间中的程序
 - 对每个程序计算概率
 - 返回概率最大的程序
-
- 能否优化这个过程？



扩展枚举算法求解程序估计问题

玲珑框架L2S（包括本部分内容+语法上的静态预分析）



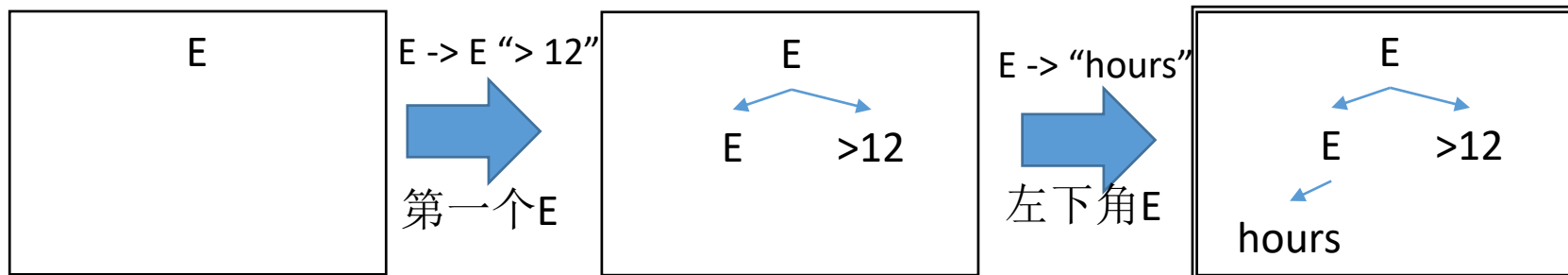
熊英飞
北京大学副教授



王博
北京交通大学讲师
北京大学博士



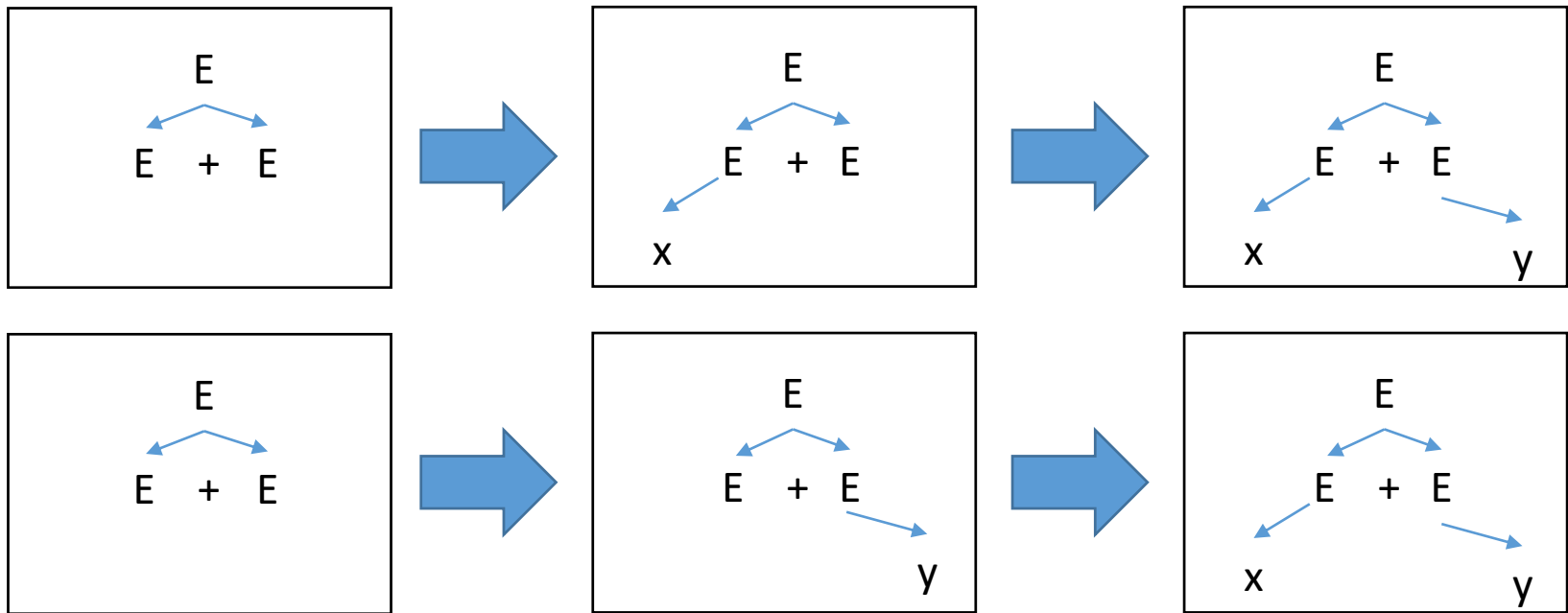
规则展开概率模型



- $P(\text{prog}) = \prod_i P(\text{position}_i \mid \text{prog}_i) P(\text{rule}_i \mid \text{prog}_i, \text{position}_i)$
 - prog_i : 当前已经展开的部分程序
 - position_i : 准备展开的终结符的位置
 - rule_i : 展开所用的规则



同一个程序，多种展开方式



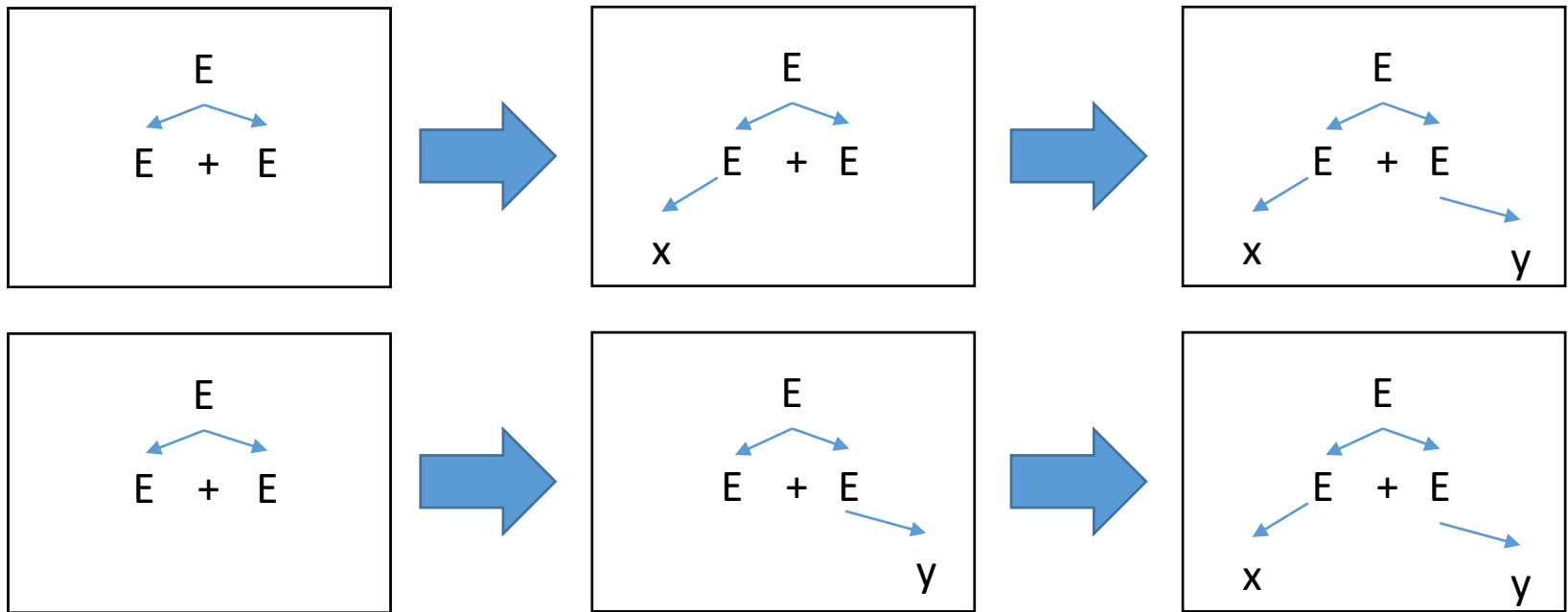


程序概率的计算

- 定理：给定任意的规则展开序列，我们有
 - $P(prog) = \prod_i P(rule_i | prog_i, position_i)$
 - $prog_i$: 第*i*步已经生成的程序
 - $position_i$: 第*i*步准备展开的非终结符的位置
 - $rule$: 第*i*步采用的产生式
 - $prog$: 完整程序



程序概率计算的收敛性



- 以上定理表明，任意展开序列都有相同概率



证明

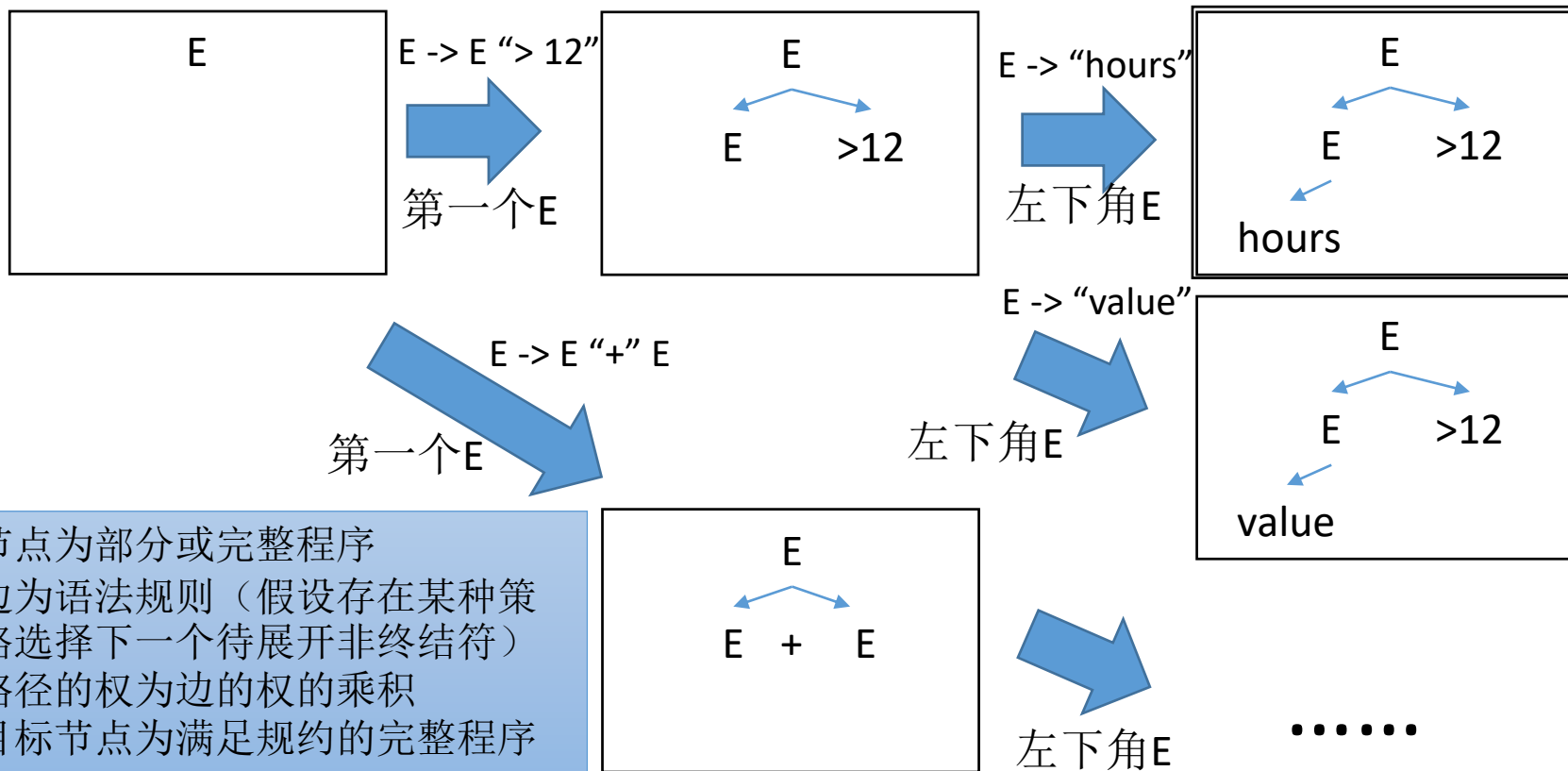
- 假设存在一个 policy，决定一个不完整程序中哪个节点先被展开，那么 policy 的选择和 prog 的概率是独立的
 - $Pr(prog)$
 - $= Pr(prog \mid policy)$ // 独立性
 - $= Pr(\langle prog_i, pos_i, rule_i \rangle_{i=1}^n \mid policy)$
 - $= Pr(prog_1 \mid policy) Pr(pos_1 \mid policy, prog_1)$
 $Pr(rule_1 \mid policy, prog_1, pos_1)$
 $Pr(eprog_2 \mid policy, prog_1, pos_1, rule_1) \dots$
 $Pr(eprog_{n+1} \mid policy, (eprog_i)_{i=1}^n, (pos_i)_{i=1}^n, (rule_i)_{i=1}^n)$
 - $= \prod_i Pr(rule_i \mid policy, (rule_j)_{j=1}^{i-1}, pos_i)$ // 删除概率为 1 的项
 - $= \prod_i Pr(rule_i \mid policy, prog_i, pos_i)$
 - $= \prod_i Pr(rule_i \mid prog_i, pos_i)$ // 独立性



规则展开概率模型的实现

- 通常计算 $P(rule_i | prog_i, position_i, context)$
 - 其中context根据需要可以为程序规约、补全的上下文等
- 可以用任意统计模型或机器学习模型实现

程序估计问题作为路径查找问题



- 节点为部分或完整程序
- 边为语法规则（假设存在某种策略选择下一个待展开非终结符）
- 路径的权为边的权的乘积
- 目标节点为满足规约的完整程序



如何求解概率最大的程序？

- 采用求解路径查找问题的标准算法
 - 迪杰斯特拉算法
 - 定向搜索（Beam Search）
 - A*算法
-
- 当概率模型预测程序满足约束的概率时，这些算法帮助避免探索概率低的程序，达到加速效果



迪杰斯特拉算法

- 定义节点的权为到达该节点的路径的最大权
- 维护一个可达节点列表，并记录每个节点的权
- 选择权最大的节点, 把该节点直接关联的新节点加入列表
- 如果某个节点已经没有未探索出边，则从列表中删除
- 反复上一步直到找到目标节点

注：在本问题中只能被一条路径到达，而在一般路径查找问题中，每个节点可以被多条路径达到，所以通用算法还需到达了旧节点时更新最大权。



迪杰斯特拉算法求解的例子

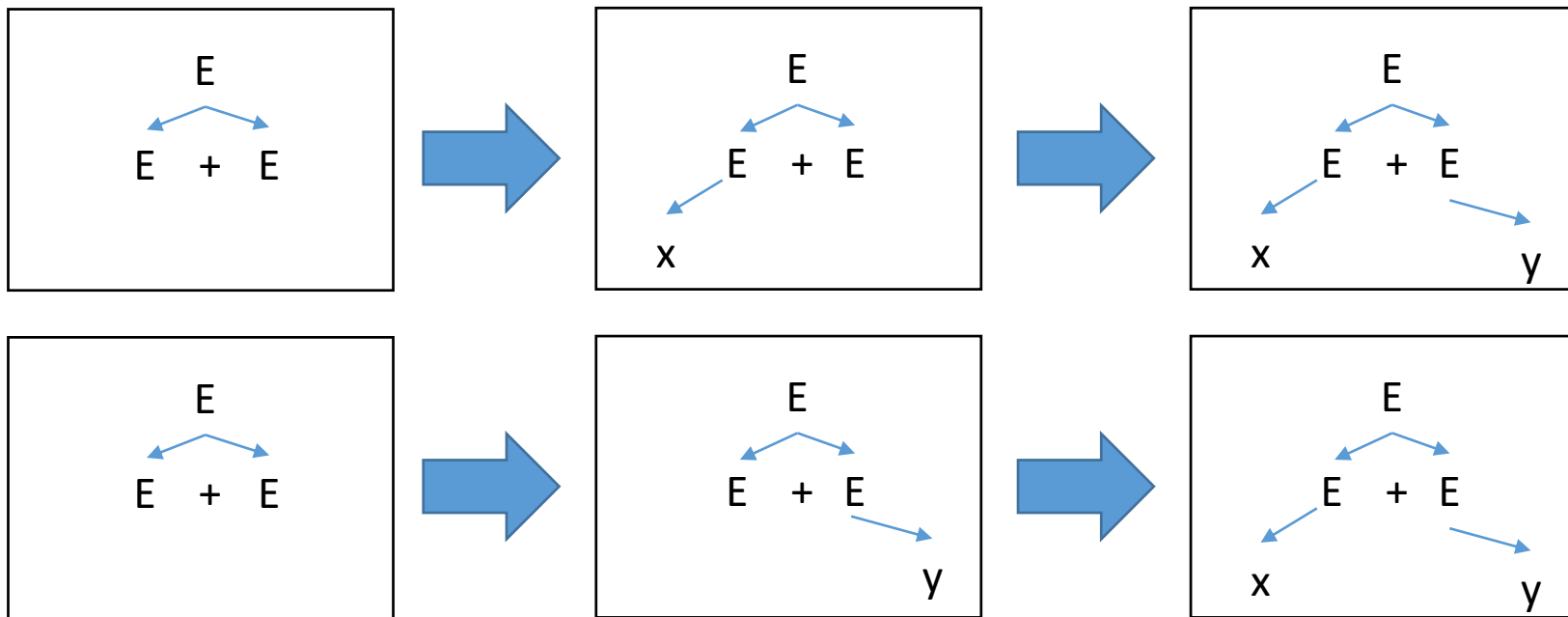
- $\langle E, 1 \rangle$
- $\langle E+E, 0.5 \rangle, \langle E-E, 0.4 \rangle, \langle \cancel{x}, 0.05 \rangle, \langle \cancel{y}, 0.05 \rangle$
- $\langle E-E, 0.4 \rangle, \langle x+E, 0.3 \rangle, \langle (E+E)+E, 0.1 \rangle, \langle y+E, 0.1 \rangle$
- $\langle x+E, 0.3 \rangle, \langle x-E, 0.2 \rangle, \langle y-E, 0.1 \rangle, \langle (E+E)+E, 0.1 \rangle, \langle y+E, 0.1 \rangle, \langle (E+E)-E, 0.05 \rangle, \langle (E-E)-E, 0.05 \rangle$
-



定义程序展开的顺序



展开的顺序

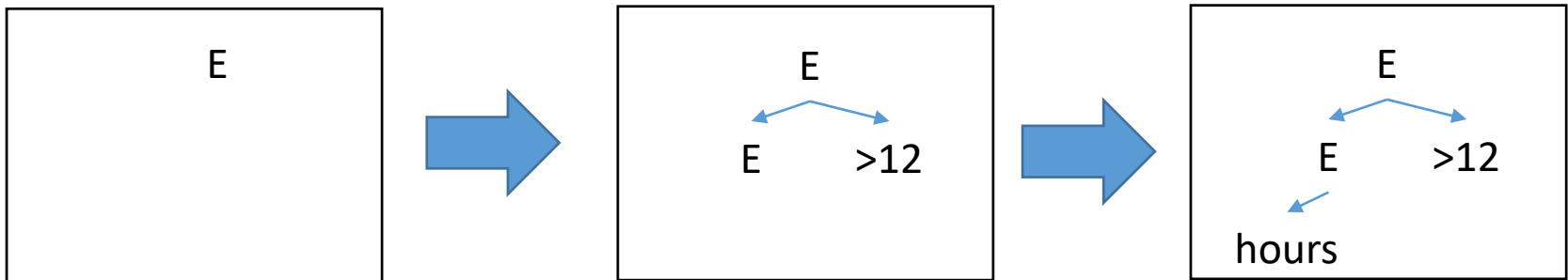


- 如果左下采用 $E \rightarrow x$ 的概率极大，而右下采用 $E \rightarrow y$ 的概率较低，则上面的顺序能显著减少搜索时间
- 需要根据应用特点定义非终结选择策略

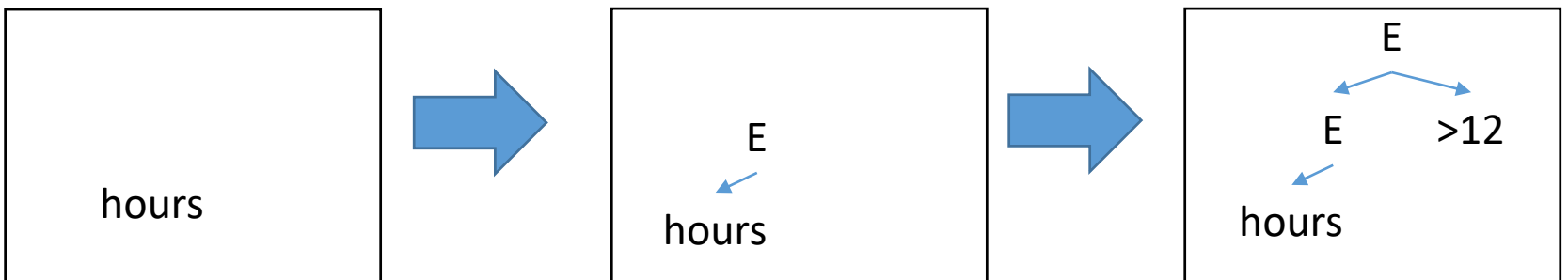


超越上下文无关文法的顺序?

- 自顶向下



- 自底向上





扩展规则

- 允许描述不同方向的语法扩展
- 由本课题组提出
- 通过采用合适的扩展规则，求解效率可提高一倍以上



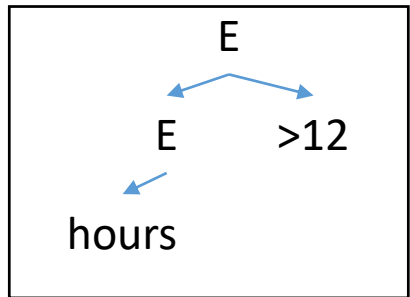
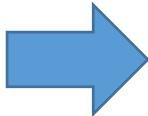
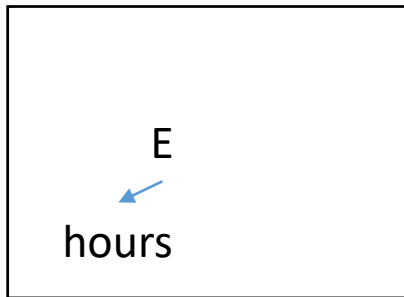
从上下文无关文法到扩展规则

$T \rightarrow E$
 $E \rightarrow E > 12 \mid E > 0 \mid E + E \mid \text{"hours"} \mid \text{"value"} \mid \dots$



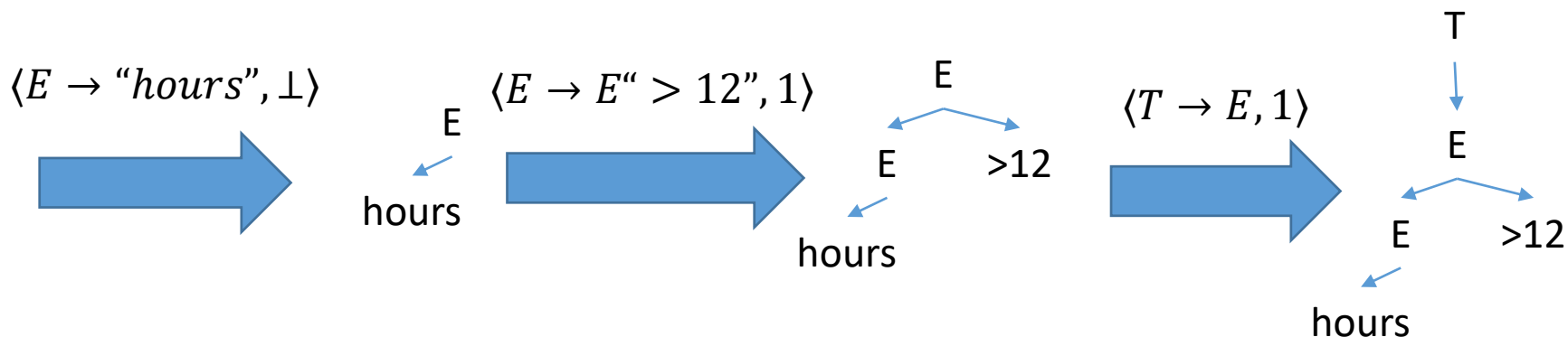
自底向上规则: $\langle E \rightarrow E > 12, 1 \rangle$
 如果第*i*个子节点已经产生, 产生整棵子树

- $\langle E \rightarrow \text{"hours"}, \perp \rangle$
- $\langle E \rightarrow \text{"value"}, \perp \rangle$
- $\langle E \rightarrow E > 12, 1 \rangle$
- $\langle E \rightarrow E + E, 1 \rangle$
- $\langle T \rightarrow E, 1 \rangle$
- $\langle E \rightarrow E > 12, 0 \rangle$
- $\langle E \rightarrow E + E, 0 \rangle$
- $\langle E \rightarrow \text{"hours"}, 0 \rangle$
- $\langle E \rightarrow \text{"value"}, 0 \rangle$



自顶向下规则: $\langle E \rightarrow E > 12, 0 \rangle$
 如果根节点已经产生, 产生整颗子树
 创建规则: $\langle E \rightarrow \text{"hours"}, \perp \rangle$
 从零产生一颗子树

基于扩展规则的程序生成过程





扩展规则树Expansion Tree

- 抽象语法树在扩展规则上的对应，记录扩展规则如何被应用的

| hours>12 | hours+value |
|------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $(T \rightarrow E, 1)$ ↑ $(E \rightarrow E \text{ " > 12" }, 1)$ ↑ $(E \rightarrow \text{"hours"}, \perp)$ | $(T \rightarrow E, 1)$ ↑ $(E \rightarrow E \text{ "+" } E, 1)$ ↙ ↘ $(E \rightarrow \text{"hours"}, \perp)$ $(E \rightarrow \text{"value"}, 0)$ |



抽象语法树 \rightarrow 扩展规则树

- 扩展规则的性质

- 完整性: 对任意AST, 至少有一个扩展规则树
- 唯一性: 对任意AST, 最多有一个扩展规则树

- 是否总是存在完整和唯一的扩展规则集合?



唯一和完整集合的充分条件

$$T \rightarrow E$$
$$E \rightarrow E \text{ " > 12" } \mid E \text{ " > 0" } \mid E \text{ " + " } E \mid \text{"hours"} \mid \text{"value"} \mid \dots$$


| | |
|-------------------------------------------------------|--|
| $\langle E \rightarrow \text{"hours"}, \perp \rangle$ | |
| $\langle E \rightarrow \text{"value"}, \perp \rangle$ | |
| $\langle E \rightarrow E \text{ " > 12"}, 1 \rangle$ | |
| $\langle E \rightarrow E \text{ " + " } E, 1 \rangle$ | |
| $\langle T \rightarrow E, 1 \rangle$ | |
| $\langle E \rightarrow E \text{ " > 12"}, 0 \rangle$ | |
| $\langle E \rightarrow E \text{ " + " } E, 0 \rangle$ | |
| $\langle E \rightarrow \text{"hours"}, 0 \rangle$ | |
| $\langle E \rightarrow \text{"value"}, 0 \rangle$ | |

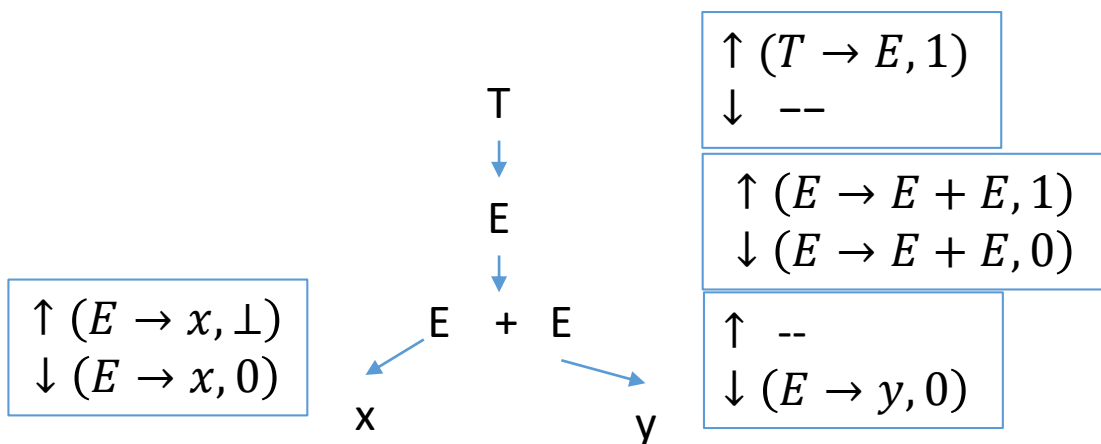
1. 除了初始符号开头的规则，所有语法规则都有对应的自顶向下展开规则
2. 所有语法规则最多只有一条自底向上的展开规则
3. 对于所有从初始符号（延自底向上展开规则）反向可达的非终结符，其所有语法规则都有一条自底向上展开规则或创建规则

从初始符号开始选择创建/自底向上规则即可



抽象语法树 -> 扩展规则树

- 利用一个动态规划算法，AST可以在 $O(n)$ 时间内转成Expansion Tree
 - 后根次序依次判断每个AST结点是否可以被自底向上和自顶向下的方式生成，如果可以，记录下采用的规则
 - 先根次序恢复出Expansion Tree





求解程序估计问题

- 给定某种结点选择策略，可以从扩展规则树得到展开序列
- 同样看做路径查找问题求解



扩展FlashMeta求解程序估计问题



FlashMeta vs 程序估计问题

- 能否采用FlashMeta求解程序估计问题?
- 方案:
 - 套入CEGIS框架得到输入输出样例
 - 首先根据输入输出样例建VSA
 - 然后将VSA作为程序空间，用玲珑框架求解概率最大的程序
 - 为便于统计，计算规则概率时忽略返回值约束
- 问题：建VSA没有被概率引导，无法加速



MaxFlash

- MaxFlash
 - 2020年由北京大学吉如一等人提出
 - 采用概率引导VSA构建
 - 效率超过FlashMeta达400-2000倍



吉如一
北京大学博士生



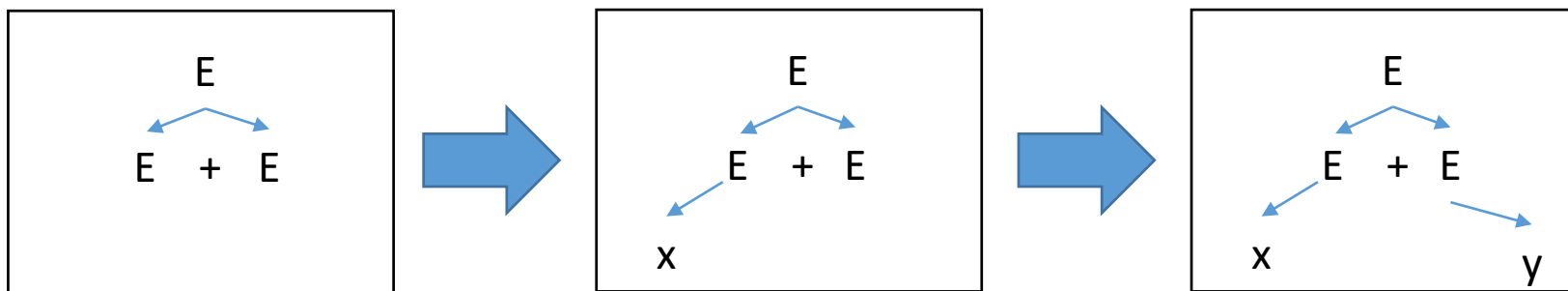
概率计算和VSA构建的矛盾

- 假如我们获得如下VSA展开式
 - $[acc]S \rightarrow [a]S + [cc]S$
- $[a]S$ 的展开式和 $[cc]S$ 的展开式是两个独立问题，可以分别求解，形成分治（和动态规划）算法
- 但 $[cc]S$ 的展开式的概率取决于 $[a]S$ 是如何展开的，无法分治
- 导致在创建VSA的时候无法应用概率引导



解决方案：自顶向下预测模型

- 节点展开规则概率只取决于其祖先，即兄弟节点的展开规则相互独立
 - 通常定义为依赖最近k层祖先节点



$$P(x + y) = P(E \rightarrow E + E \mid \perp)P(E \rightarrow x \mid E)P(E \rightarrow y \mid E)$$



统一概率计算和VSA构建

| 带祖先的VSA | 产生式概率 | 最优程序和概率 |
|---------------------------------------------------|-------|------------|
| $[acc \perp]S \rightarrow [a S]S + [cc S]S$ | 0.9 | x+y, 0.081 |
| $ [ac S]S + [c S]S$ | 0.9 | |
| $[ac S]S \rightarrow [a S]S + [c S]S$ | 0.1 | x+z, 0.009 |
| $[cc S]S \rightarrow [c S]S + [c S]S$ | 0.1 | y, 0.3 |
| $ y$ | 0.3 | |
| $[a S]S \rightarrow x$ | 0.3 | x, 0.3 |
| $[c S]S \rightarrow z$ | 0.3 | z, 0.3 |

可采用动态规划算法独立求解每个子问题



基于概率的剪枝：分支限界

- 假设我们认为最优程序的概率应大于0.3

| 带祖先和概率下界的VSA | 概率 | 说明 |
|-----------------------------------------------------------------------|-----|----------------|
| $[acc \perp 0.3]S \rightarrow [a S 0.33]S + [cc S 0.33]S$ | 0.9 | $0.3/0.9=0.33$ |
| $[cc S 0.33]S \rightarrow [c S 3.3]S + [c S 3.3]S$ | 0.1 | $0.33/0.1=3.3$ |
| ψ | 0.3 | |



基于概率的剪枝：估价函数

- 静态分析非终结符的概率上界

| 祖先 | 非终结符 | 概率上界 |
|----|------|-------|
| L | S | 0.081 |
| S | S | 0.3 |

- 假设我们认为最优程序的概率应大于0.3

| 带祖先和概率下界的VSA | 概率 | 说明 |
|-------------------------------------------------------------------|-----|--------------------|
| $[acc L 0.3]S \rightarrow [a S 1.11]S + [cc S 1.11]S$ | 0.9 | $0.3/0.9/0.3=1.11$ |



基于概率的剪枝：迭代加深

- 如何知道最优程序的概率应大于多少？
 - 设置一个概率下界，并逐步放宽
 - 如，一开始是0.1，然后每次除以10

基于概率的剪枝：复用于子问题



- 概率下界基本不可能相同
 - 动态规划退化成分治
- 需要复用概率下界不同的子问题
 - 考虑两个除了概率下界不同以外，其他都一样的子问题 $(P, 0.2)$, $(P, 0.1)$
 - Case 1: $(P, 0.2)$ 先于 $(P, 0.1)$
 - 有解，则同样是 $(P, 0.1)$ 的解；
 - 无解，则可以更新 P 的估价函数
 - Case 2: $(P, 0.1)$ 先于 $(P, 0.2)$
 - 有解，则同样是 $(P, 0.2)$ 的解（因为总是搜索概率最大的结果）
 - 无解， $(P, 0.2)$ 同样无解



应用效果



玲珑框架应用

——从自然语言生成代码

- 已有方法主要采用 end-to-end 的神经网络结构，如 RNN, LSTM
- RNN/LSTM 有长依赖问题
 - 长依赖问题 Long dependency problem: 无法处理距离较远的依赖关系
 - 程序中长依赖很多，如当前使用的变量可能很早之前声明



```
[NAME]
Acidic Swamp Ooze
[ATK] 3
[DEF] 2
[COST] 2
[DUR] -1
[TYPE] Minion
[CLASS] Neutral
[RACE] NIL
[RARITY] Common
[DESCRIPTION]
"Battlecry: Destroy Your Opponent's Weapon"
```



```
class AcidicSwampOoze(MinionCard):
    def __init__(self):
        super().__init__("Acidic Swamp Ooze", 2,
            CHARACTER_CLASS.ALL, CARD_RARITY.COMMON,
            battlecry=Battlecry(Destroy(), WeaponSelector(EnemyPlayer())))

    def create_minion(self, player):
        return Minion(3, 2)
```



结果1[AAAI19]

玲珑框架允许我们采用任意分类模型
采用长依赖问题较小的CNN

Benchmark: HearthStone

| Model | StrAcc | Acc+ | BLEU |
|---------------------------------------------------|---------------|-------------|-------------|
| LPN (Ling et al. 2016) | 6.1 | – | 67.1 |
| SEQ2TREE (Dong and Lapata 2016) | 1.5 | – | 53.4 |
| SNM (Yin and Neubig 2017) | 16.2 | ~18.2 | 75.8 |
| ASN (Rabinovich, Stern, and Klein 2017) | 18.2 | – | 77.6 |
| ASN+SUPATT (Rabinovich, Stern, and Klein 2017) | 22.7 | – | 79.2 |
| Our system | 27.3 | 30.3 | 79.6 |



结果2[AAAI20]

- 将CNN换成Transformer
 - Transformer: 2017年新提出来的网络体系结构
 - L2S允许灵活替换不同的统计模型

| | Model | StrAcc | Acc+ | BLEU |
|------------|--------------------------------------|-------------|-------------|-------------|
| Plain | LPN (Ling et al., 2016) | 6.1 | – | 67.1 |
| | SEQ2TREE (Dong and Lapata, 2016) | 1.5 | – | 53.4 |
| | YN17 (Yin and Neubig, 2017) | 16.2 | ~18.2 | 75.8 |
| | ASN (Rabinovich et al., 2017) | 18.2 | – | 77.6 |
| | ReCode (Hayati et al., 2018) | 19.6 | – | 78.4 |
| | CodeTrans-A | 25.8 | 25.8 | 79.3 |
| Structured | ASN+SUPATT (Rabinovich et al., 2017) | 22.7 | – | 79.2 |
| | SZM19 (Sun et al., 2019) | 27.3 | 30.3 | 79.6 |
| | CodeTrans-B | 31.8 | 33.3 | 80.8 |

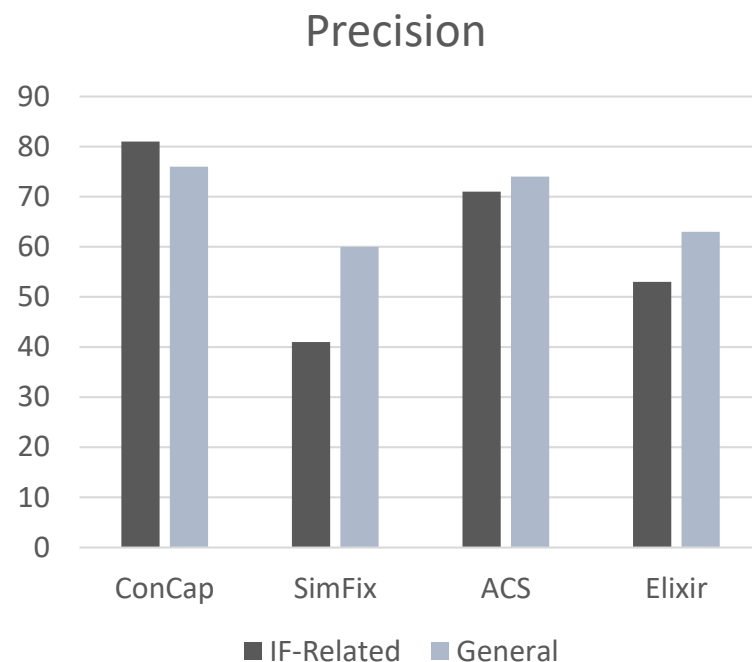
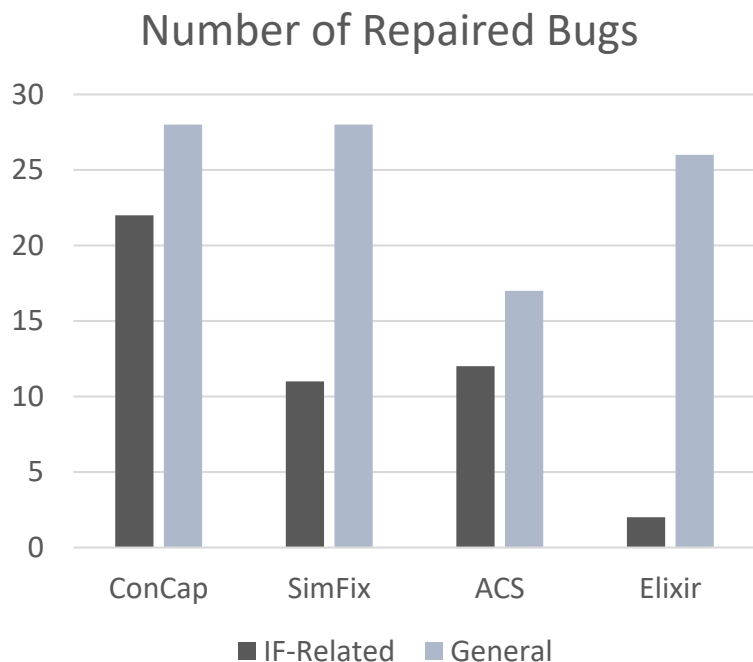
玲珑框架应用

——修复条件缺陷 [TOSEM投稿]



采用xgboost和类型语义约束来解决条件合成问题

Benchmark: Defects4J



条件缺陷修复数量和准确率达到最高
8个没有被任何别的工作修复过的全新缺陷

缺陷修复最新结果[ESEC/FSE21]



采用神经网络来构造概率模型，并用修改操作定义了程序空间

Table 2: Comparison without Perfect Fault Localization

| Project | jGenProg | HDRepair | Nopol | CapGen | SketchFix | FixMiner | SimFix | TBar | DLFix | PraPR | AVATAR | Recoder |
|---------|----------|----------|-------|-------------|-----------|----------|-------------|--------------|-------|--------|------------|--------------|
| Chart | 0/7 | 0/2 | 1/6 | 4/4 | 6/8 | 5/8 | 4/8 | 9/14 | 5/12 | 4/14 | 5/12 | 8/14 |
| Closure | 0/0 | 0/7 | 0/0 | 0/0 | 3/5 | 5/5 | 6/8 | 8/12 | 6/10 | 12/62 | 8/12 | 17/31 |
| Lang | 0/0 | 2/6 | 3/7 | 5/5 | 3/4 | 2/3 | 9/13 | 5/14 | 5/12 | 3/19 | 5/11 | 9/15 |
| Math | 5/18 | 4/7 | 1/21 | 12/16 | 7/8 | 12/14 | 14/26 | 18/36 | 12/28 | 6/40 | 6/13 | 15/30 |
| Time | 0/2 | 0/1 | 0/1 | 0/0 | 0/1 | 1/1 | 1/1 | 1/3 | 1/2 | 0/7 | 1/3 | 2/2 |
| Mockito | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 0/0 | 1/2 | 1/1 | 1/6 | 2/2 | 2/2 |
| Total | 5/27 | 6/23 | 5/35 | 21/25 | 19/26 | 25/31 | 34/56 | 42/81 | 30/65 | 26/148 | 27/53 | 55/94 |
| P(%) | 18.5 | 26.1 | 14.3 | 84.0 | 73.1 | 80.6 | 60.7 | 51.9 | 46.2 | 17.6 | 50.9 | 56.4 |

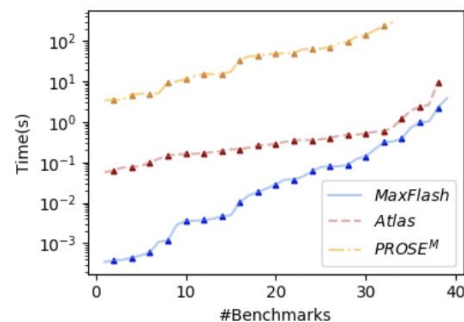
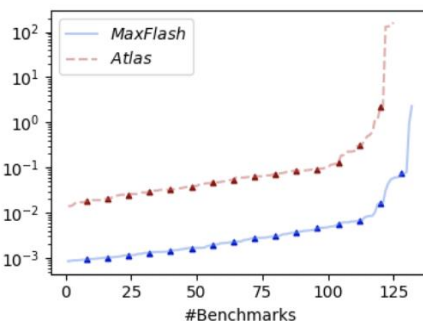
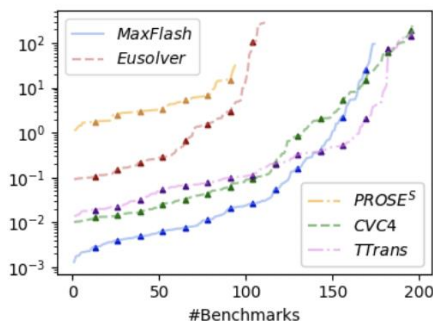
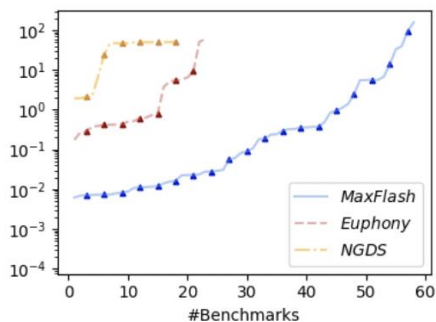
In the cells, x/y:x denotes the number of correct patches, and y denotes the number of patches that can pass all the test cases.

神经网络修复首次超过传统修复的效果
被审稿人提名最佳论文候选

加速传统程序合成[OOPSLA20]



- 相比于已有的程序合成技术，MaxFlash
 - (1)取得了 $\times 4$ - $\times 2080$ 倍的平均加速比；
 - (2)在500ms的响应时间内解决了更多的合成任务；
 - (3)更加节约空间。





参考资料

- Syntax-Guided Synthesis. R. Alur, R. Bodik, G. Juniwal, P. Madusudan, M. Martin, M. Raghothman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak and A. Udupa. In 13th International Conference on Formal Methods in Computer-Aided Design, 2013.
- Sumit Gulwani, Oleksandr Polozov, Rishabh Singh: Program Synthesis. Foundations and Trends in Programming Languages 4(1-2): 1-119 (2017)
- Polozov O , Gulwani S . FlashMeta: a framework for inductive program synthesis[C]// Acm Sigplan International Conference on Object-oriented Programming. ACM, 2015.



参考文献

- Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, Ashish Tiwari: Oracle-guided component-based program synthesis. ICSE (1) 2010: 215-224
- Xinyu Wang, Isil Dillig, and Rishabh Singh。 Synthesis of Data Completion Scripts using Finite Tree Automata. OOPSLA, 2017
- Feng Y , Martins R , Bastani O , et al. Program Synthesis using Conflict-Driven Learning[J]. PLDI 2018.



参考文献

- Yingfei Xiong, Bo Wang, Guirong Fu, Linfei Zang. Learning to Synthesize. GI'18: Genetic Improvement Workshop, May 2018.
- Yingfei Xiong, Bo Wang. L2S: a Framework for Synthesizing the Most Probable Program under a Specification. Arxiv, 2021.
- Ruyi Ji, Yican Sun, Yingfei Xiong, Zhenjiang Hu. Guiding Dynamic Programming via Structural Probability for Accelerating Programming by Example. OOPSLA'20: Object-Oriented Programming, Systems, Languages, and Applications 2020, November 2020.



参考文献

- Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, Lu Zhang. A Grammar-Based Structural CNN Decoder for Code Generation. AAIL'19: Thirty-Third AAIL Conference on Artificial Intelligence, January 2019.
- Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, Lu Zhang. TreeGen: A Tree-Based Transformer Architecture for Code Generation. AAIL'20: Thirty-Fourth AAIL Conference on Artificial Intelligence, January 2020.
- Qihao Zhu, Zeyu Sun, Yuanan Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, Lu Zhang. A Syntax-Guided Edit Decoder for Neural Program Repair. ESEC/FSE'21: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, August 2021.