



软件分析

# 程序合成：概率

熊英飞

北京大学



# 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



# 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



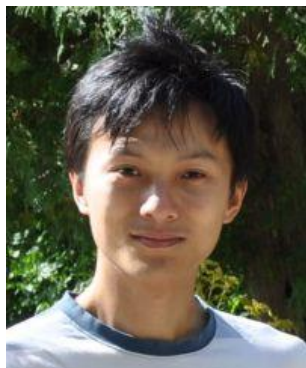
# 基本算法：穷举

- 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）



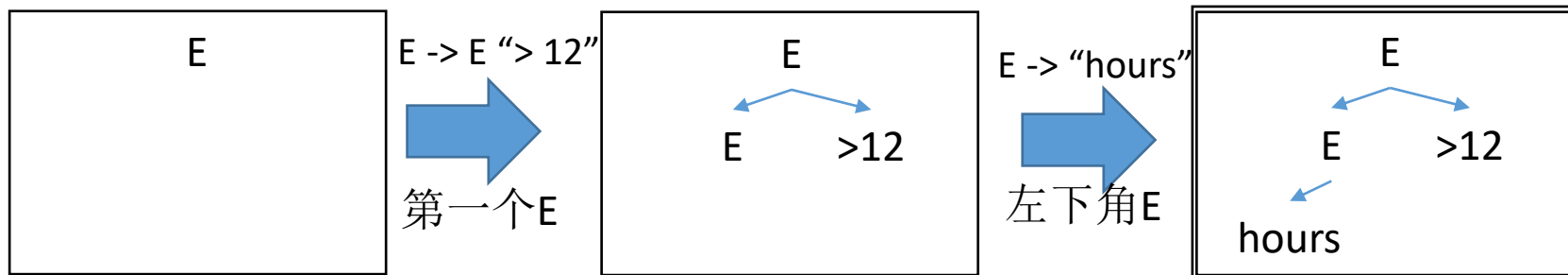
熊英飞  
北京大学副教授



王博  
北京交通大学讲师  
北京大学博士



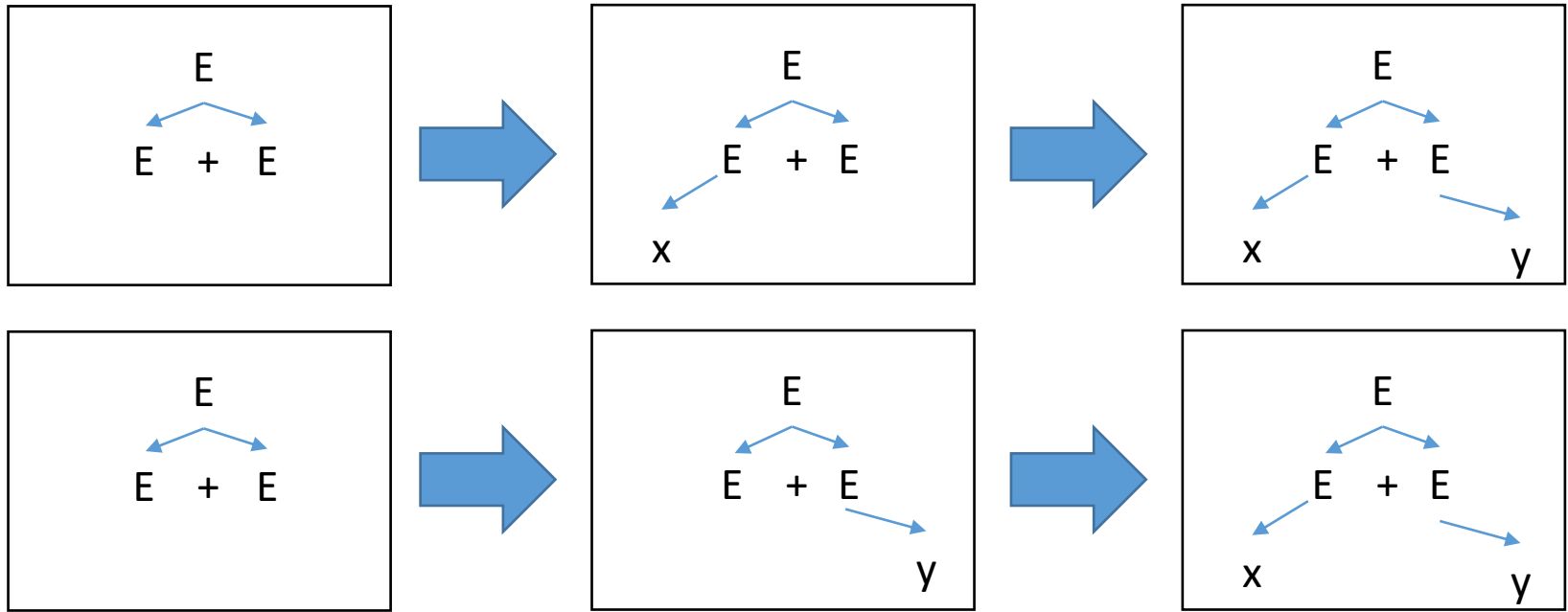
# 规则展开概率模型



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



# 同一个程序，多种展开方式





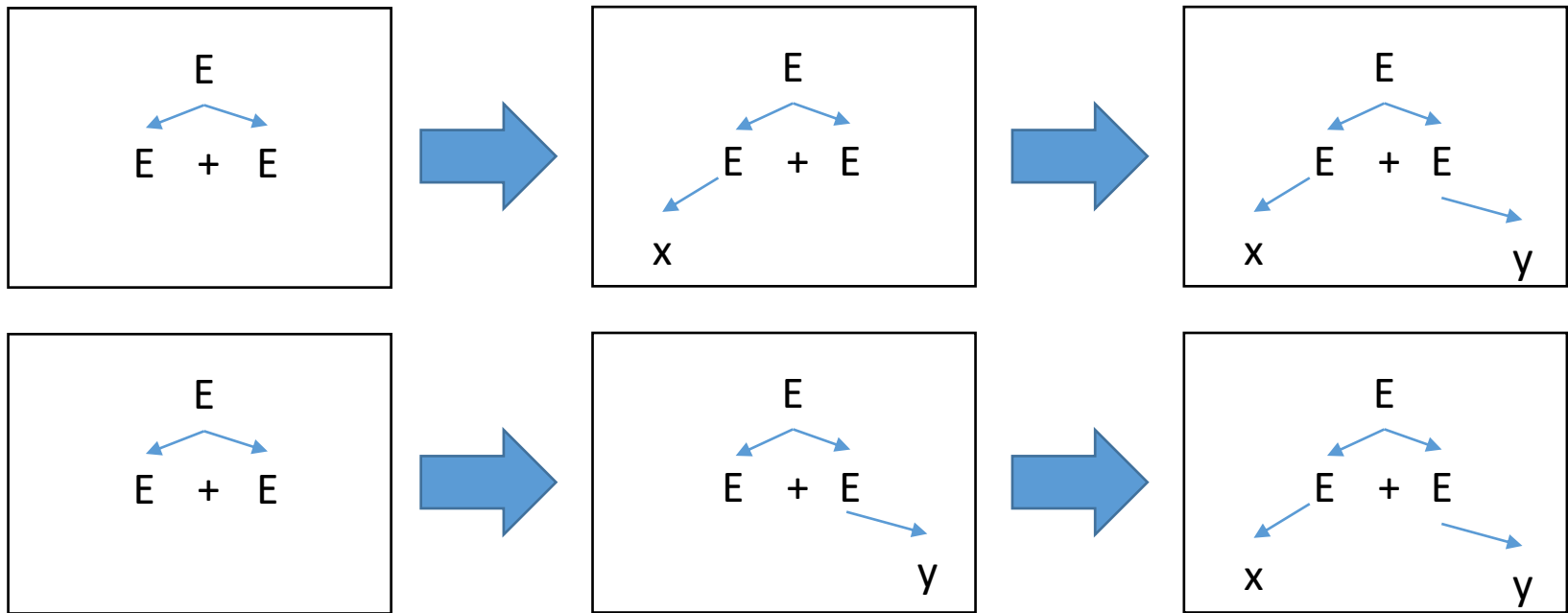


# 程序概率的计算

- 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

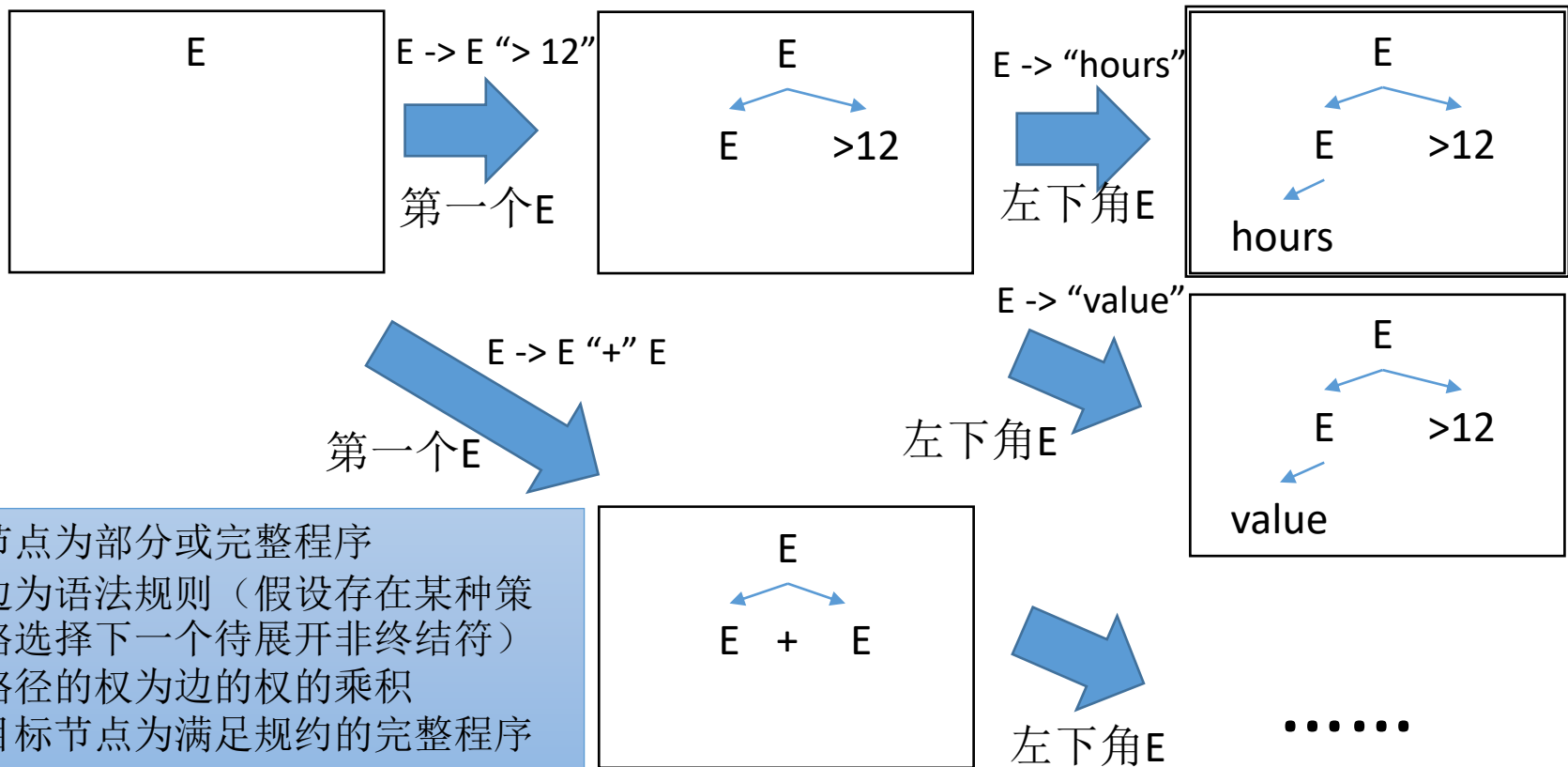
- 假设存在一个 policy，决定一个不完整程序中哪个节点先被展开，那么 policy 的选择和 prog 的概率是独立的
  - $Pr(prog)$
  - $= Pr(prog \mid policy)$  // 独立性
  - $= Pr(\langle prog_i, pos_i, rule_i \rangle_{i=1}^n \mid policy)$
  - $= Pr(prog_1 \mid policy) Pr(pos_1 \mid policy, prog_1)$   
 $Pr(rule_1 \mid policy, prog_1, pos_1)$   
 $Pr(eprog_2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

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





# 定向搜索 (Beam Search)

- 在迪杰斯特拉算法中不保留所有节点，只保留概率最大的k个
- 近似算法，不保证最优，也不保证找到结果



# A\*算法

- 节点n的权=到达该节点的权\*h(n)
  - $h(n)$ =剩余路径权的上界
- 其他同迪杰斯特拉算法
- 如何知道剩余路径权的上界?
  - 假设存在函数 $\hat{P}(rule)$ ，满足
    - $\forall prog, position: \hat{P}(rule) \geq P(rule | prog, position)$
  - 在语法展开式上做静态分析，分析出每个非终结符的概率上界
    - 从  $E \rightarrow E + E \mid x \mid y \mid \dots$
    - 得到方程  $\hat{P}(E) = \max(\hat{P}(E \rightarrow E + E)\hat{P}(E)\hat{P}(E), \hat{P}(E \rightarrow x), \hat{P}(E \rightarrow y), \dots)$
  - 剩余路径权的上界为所有未展开非终结符概率上界的积



# 剪枝

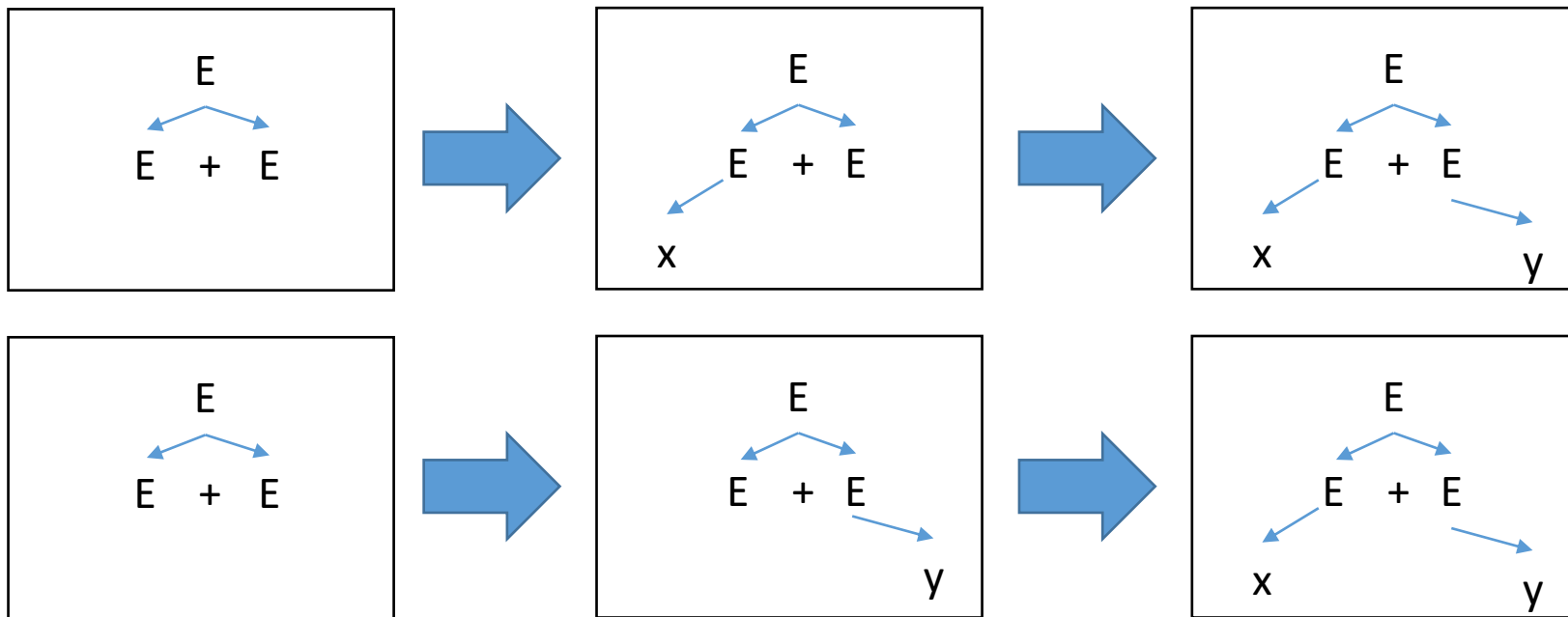
- 之前描述的剪枝过程仍然可以用于求解程序估计问题
- 判断出一个部分程序无法满足规约时，从列表中移除对应节点



# 定义程序展开的顺序



# 展开的顺序

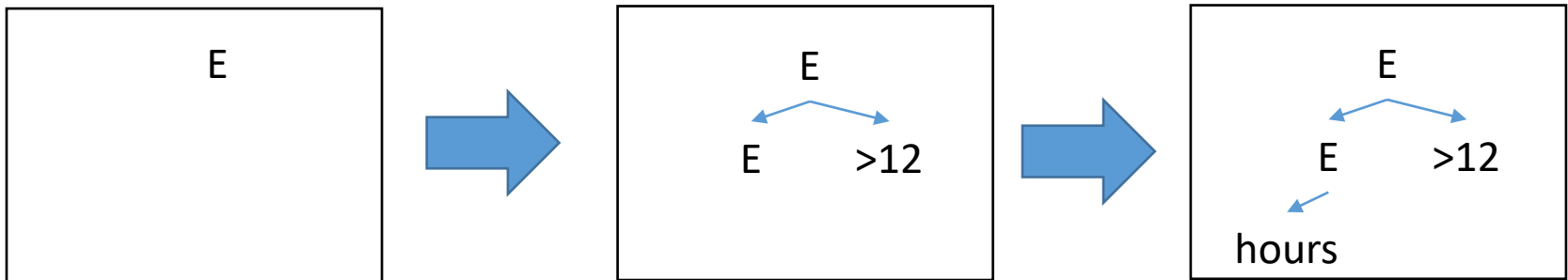


- 如果左下采用 $E \rightarrow x$ 的概率极大，而右下采用 $E \rightarrow y$ 的概率较低，则上面的顺序能显著减少搜索时间
- 需要根据应用特点定义非终结选择策略

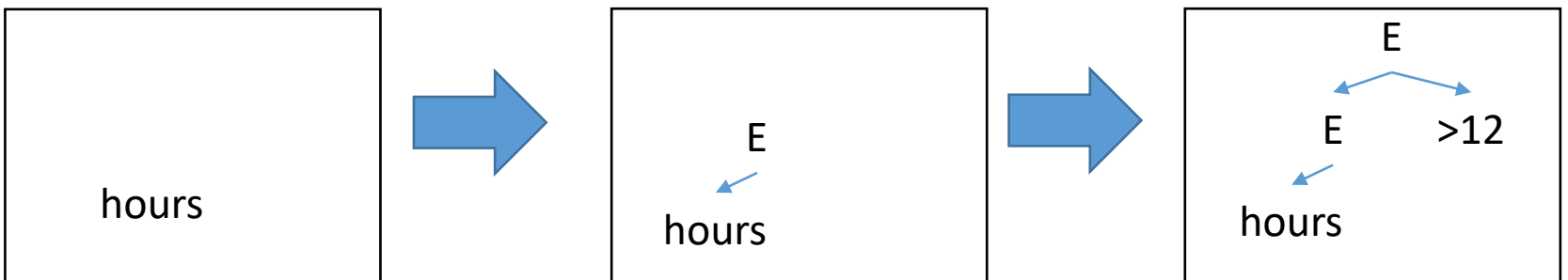


# 超越上下文无关文法的顺序?

- 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



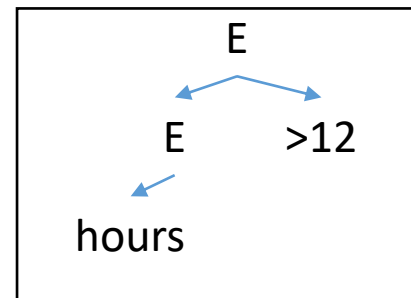
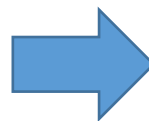
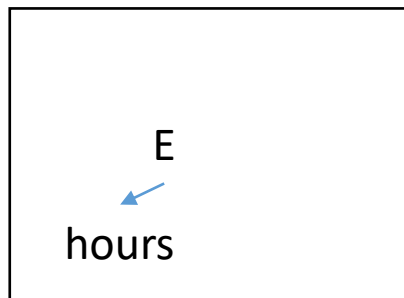
# 从上下文无关文法到扩展规则

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



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

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

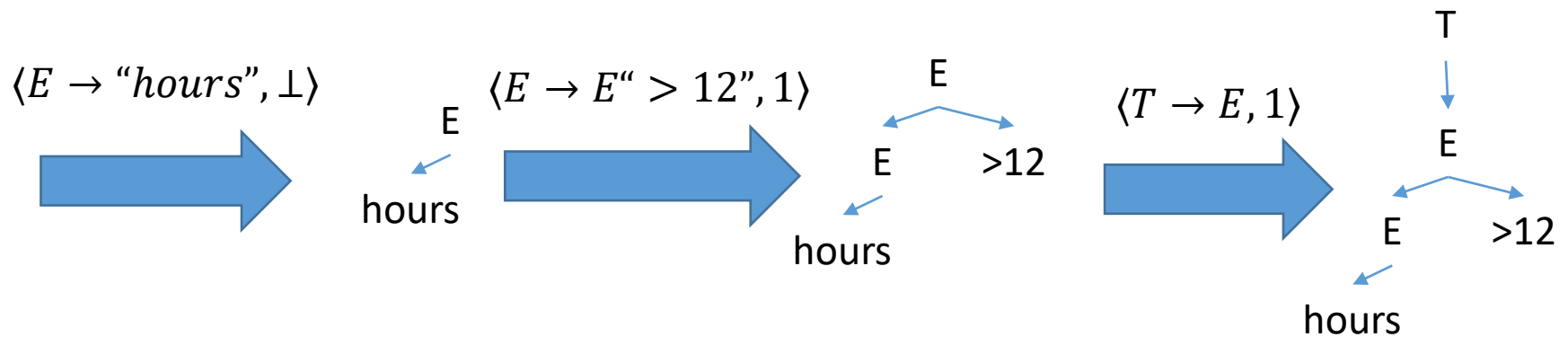


自顶向下规则:  $\langle E \rightarrow E > 12, 0 \rangle$   
 如果根节点已经产生, 产生整颗子树  
 创建规则:  $\langle E \rightarrow \text{"hours"}, \perp \rangle$   
 从零产生一颗子树





# 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

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


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

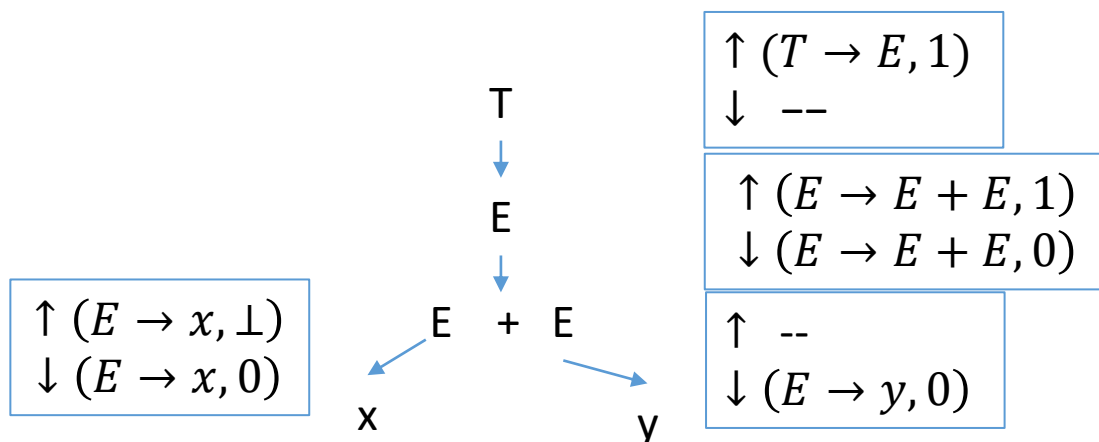
1. 除了初始符号开头的规则，所有语法规则都有对应的自顶向下展开规则
2. 所有语法规则最多只有一条自底向上的展开规则
3. 对于所有从初始符号（延自底向上展开规则）反向可达的非终结符，其所有语法规则都有一条自底向上展开规则或创建规则

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



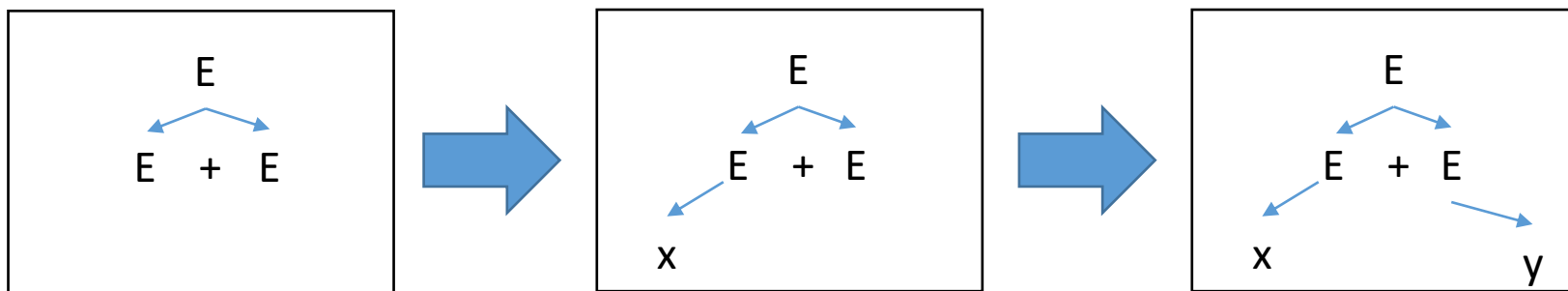
# 概率计算和VSA构建的矛盾

- 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



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



# 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	
$  [ac   S]S + [c   S]S$	0.9	

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	$0.33/0.1=3.3$
$  \psi$	0.3	$0.33/0.3=1.1$



# 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

<b>Model</b>	<b>StrAcc</b>	<b>Acc+</b>	<b>BLEU</b>
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
<b>Our system</b>	<b>27.3</b>	<b>30.3</b>	<b>79.6</b>



# 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
	<b>CodeTrans-A</b>	<b>25.8</b>	<b>25.8</b>	<b>79.3</b>
Structured	ASN+SUPATT (Rabinovich et al., 2017)	22.7	–	79.2
	SZM19 (Sun et al., 2019)	27.3	30.3	79.6
	<b>CodeTrans-B</b>	<b>31.8</b>	<b>33.3</b>	<b>80.8</b>

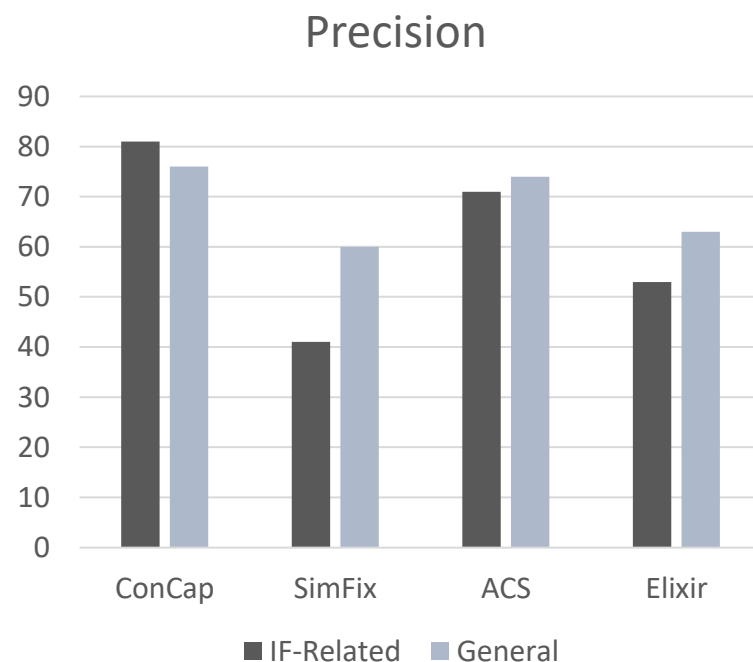
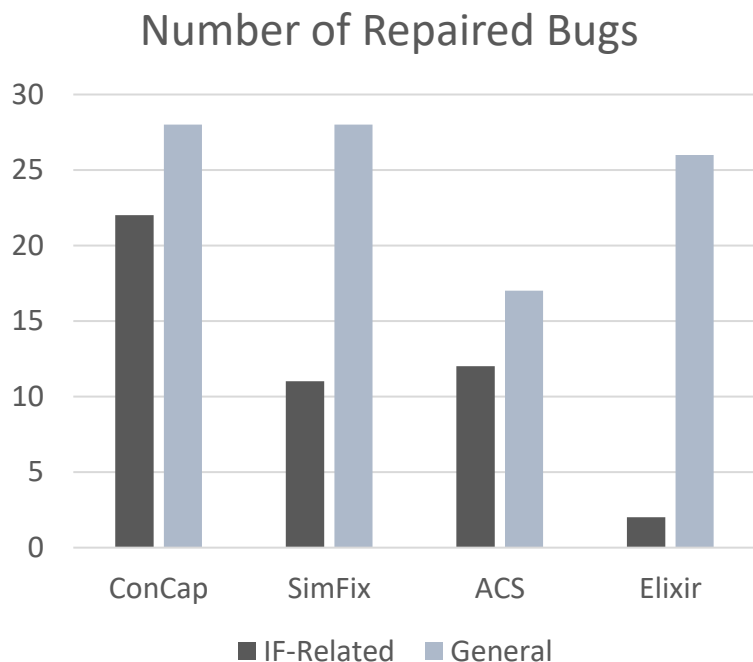
# 玲珑框架应用

## ——修复条件缺陷 [TOSEM]



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	<b>9/14</b>	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	<b>17/31</b>
Lang	0/0	2/6	3/7	5/5	3/4	2/3	<b>9/13</b>	5/14	5/12	3/19	5/11	<b>9/15</b>
Math	5/18	4/7	1/21	12/16	7/8	12/14	14/26	<b>18/36</b>	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	<b>2/2</b>
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/2	1/1	1/6	<b>2/2</b>	<b>2/2</b>
Total	5/27	6/23	5/35	21/25	19/26	25/31	34/56	42/81	30/65	26/148	27/53	<b>53/94</b>
P(%)	18.5	26.1	14.3	<b>84.0</b>	73.1	80.6	60.7	51.9	46.2	17.6	50.9	56.4

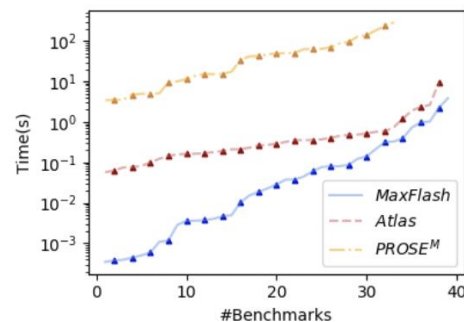
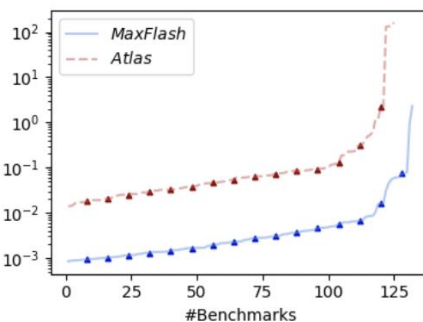
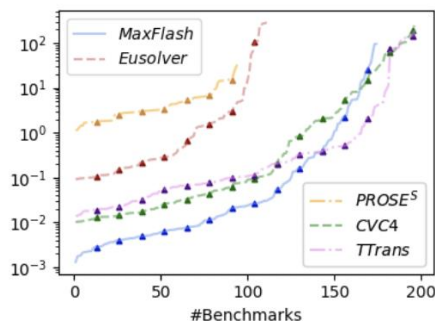
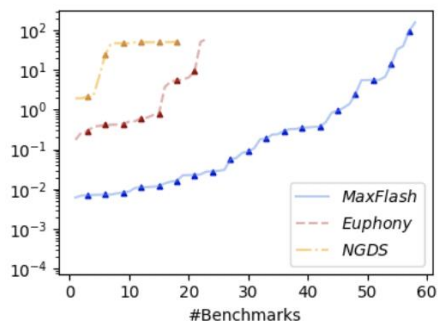
In the cells, x/y:x denotes the number of correct patches, and y denotes the number of patches that can pass all the test cases.

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. ACM Transactions on Software Engineering Methodology, Online First, Dec 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. AACL'19: Thirty-Third AACL 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. AACL'20: Thirty-Fourth AACL 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.