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


```python
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

```java
/** 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

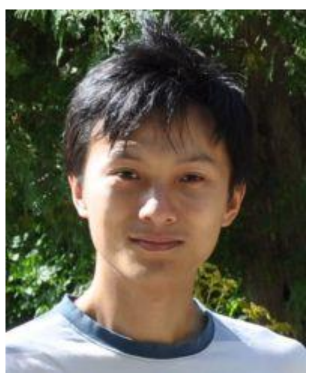
基本算法：穷举

- 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

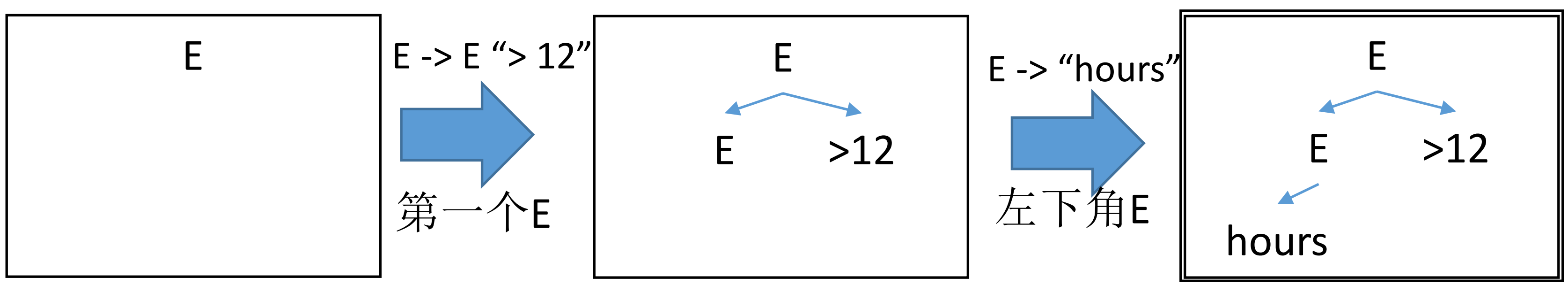
玲珑框架L2S（包括本部分内容+语法上的静态预分析）



熊英飞
北京大学副教授

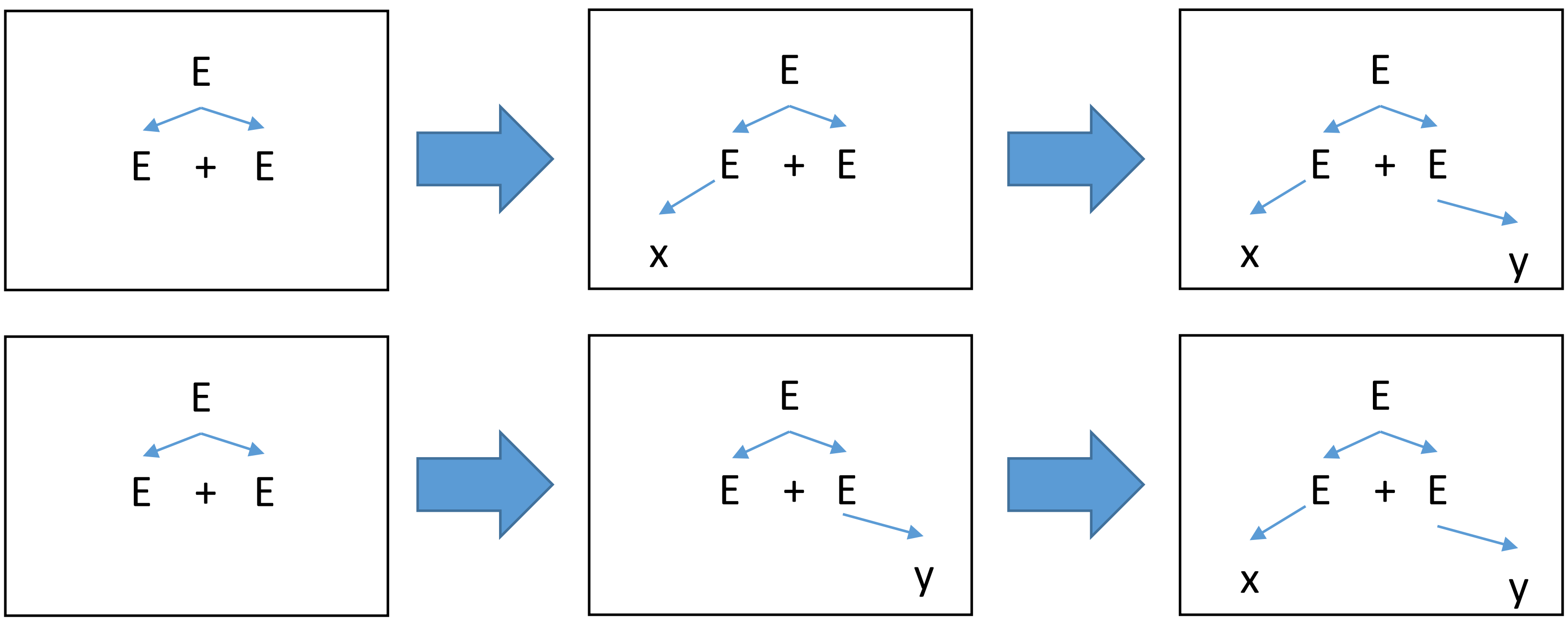
王博
北京交通大学讲师
北京大学博士

规则展开概率模型



- $P(prog) = \prod_i P(\ position_i \mid prog_i\ )P(\ rule_i \mid prog_i, position_i\ )$
  - $prog_i$: 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

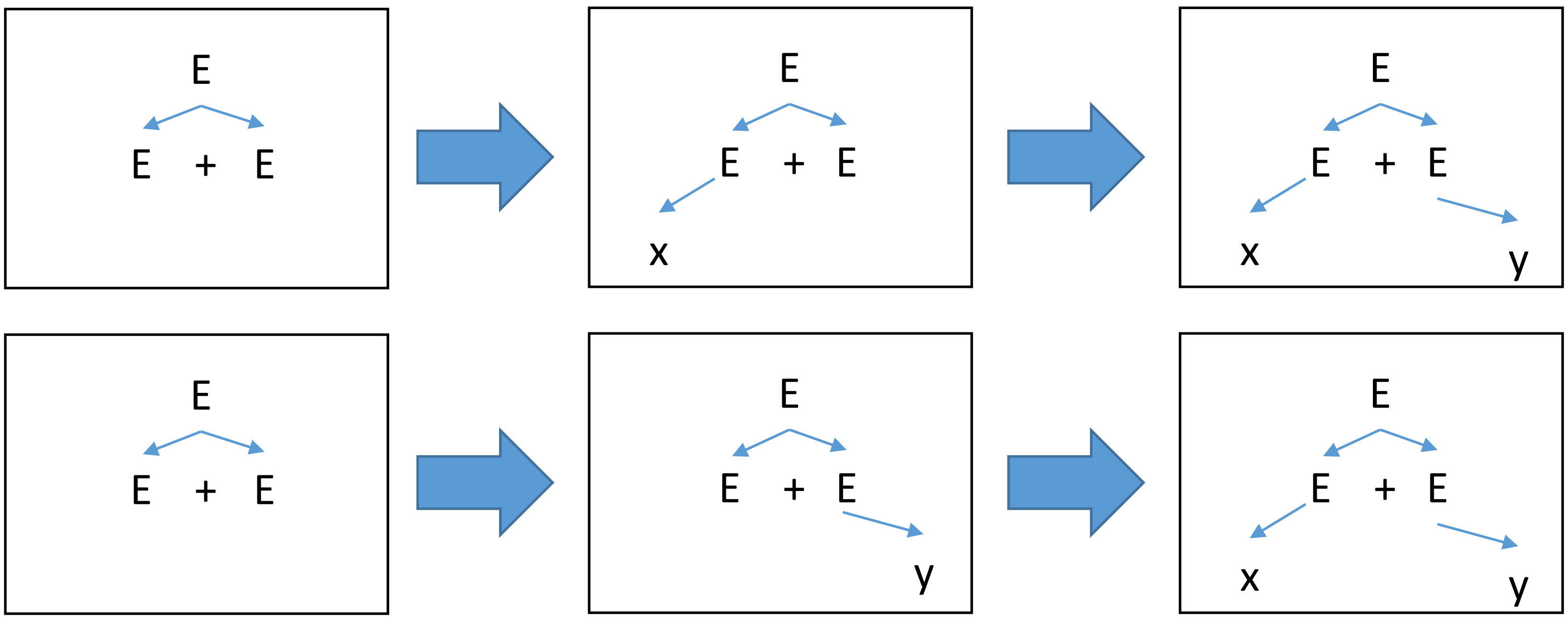


程序概率的计算

- 定理：给定任意的规则展开序列，我们有
  - $P(prog) = \prod_i P(\ rule_i \mid prog_i, position_i\ )$

  - $prog_i$: 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

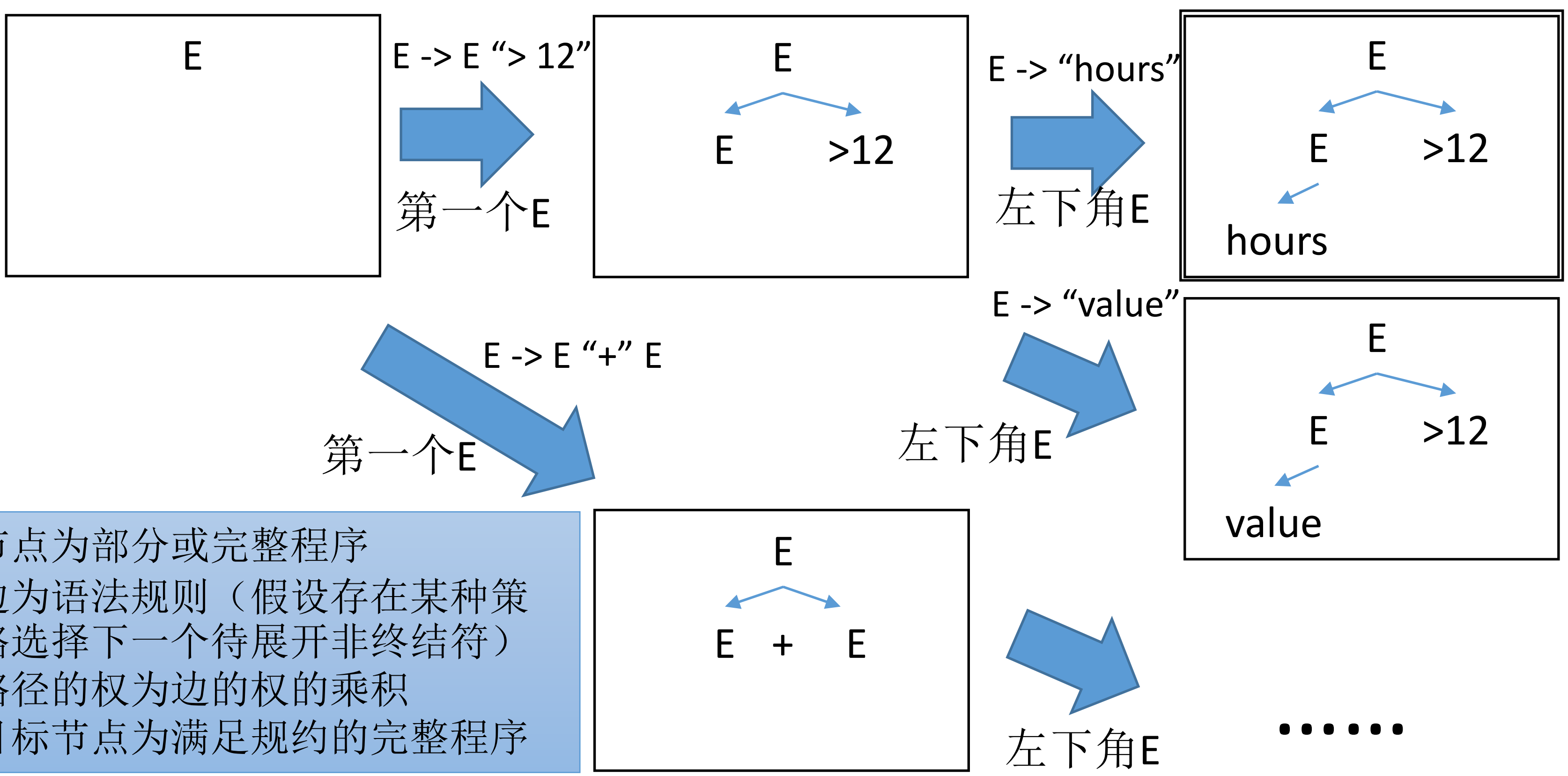
证明

- 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
  - $Pr(prog)$
  - $= \ \mathrm{Pr}(prog \mid policy)$ //独立性
  - $= Pr((\langle prog_i, pos_i, rule_i \rangle)_{i=1}^{n} \mid policy)$
  - $= Pr(prog_1 \mid policy)\, Pr(pos_1 \mid policy, prog_1)$ $Pr(rule_1 \mid policy, prog_1, pos_1)$ $Pr(eprog_2 \mid policy, prog_1, pos_1, rule_1) \ldots$ $Pr(eprog_{n+1} \mid policy, (eprog_i)_{i=1}^{n}, (pos_i)_{i=1}^{n}, (rule_i)_{i=1}^{n})$
  - $= \prod_i Pr\left(rule_i \mid policy, (rule_j)_{j=1}^{i-1}, pos_i\right)$ //删除概率为1的项
  - $= \prod_i \mathrm{Pr}(rule_i \mid policy, prog_i, pos_i)$
  - $= \prod_i \mathrm{Pr}(rule_i \mid prog_i, pos_i)$ //独立性

规则展开概率模型的实现

- 通常计算$P(\ rule_i \mid prog_i, position_i, context\ )$
  - 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

- <E,1>
- <E+E,0.5>, <E-E, 0.4>, <x, 0.05>, <y, 0.05>
- <E-E, 0.4>, <x+E, 0.3>, <(E+E)+E, 0.1>, <y+E, 0.1>, <x, 0.05>, <y, 0.05>
- <x+E, 0.3>, <x-E, 0.2>, <y-E, 0.1>, <(E+E)+E, 0.1>, <y+E, 0.1>, , <x, 0.05>, <y, 0.05>, <(E+E)-E, 0.05>, <(E-E)-E, 0.05>
- ……

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

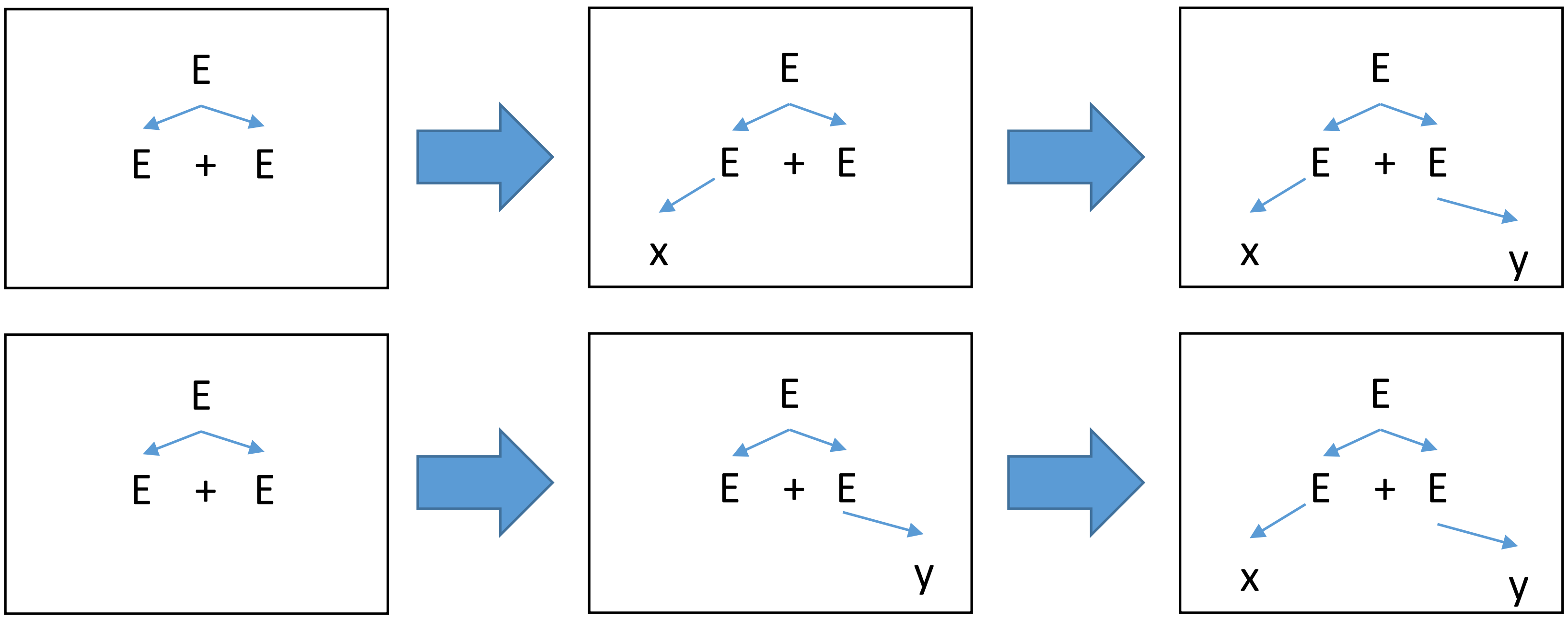
- 节点n的权=到达该节点的权*h(n)
  - h(n)=剩余路径权的上界
- 其他同迪杰斯特拉算法
- 如何知道剩余路径权的上界?
  - 假设存在函数$\hat{P}(rule)$，满足
    - $\forall prog, position: \hat{P}(rule) \geq P(\ rule \mid \ prog, position\ )$
  - 在语法展开式上做静态分析，分析出每个非终结符的概率上界
    - 从　　　　E->E+E | x | y |…
    - 得到方程　　$\hat{P}(E) = \max(\hat{P}(E \to E + E)\hat{P}(E)\hat{P}(E),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

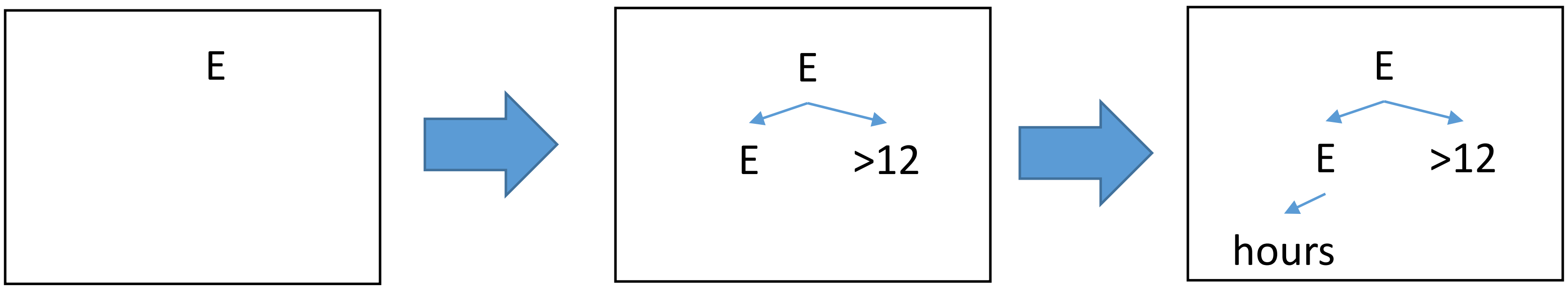
展开的顺序



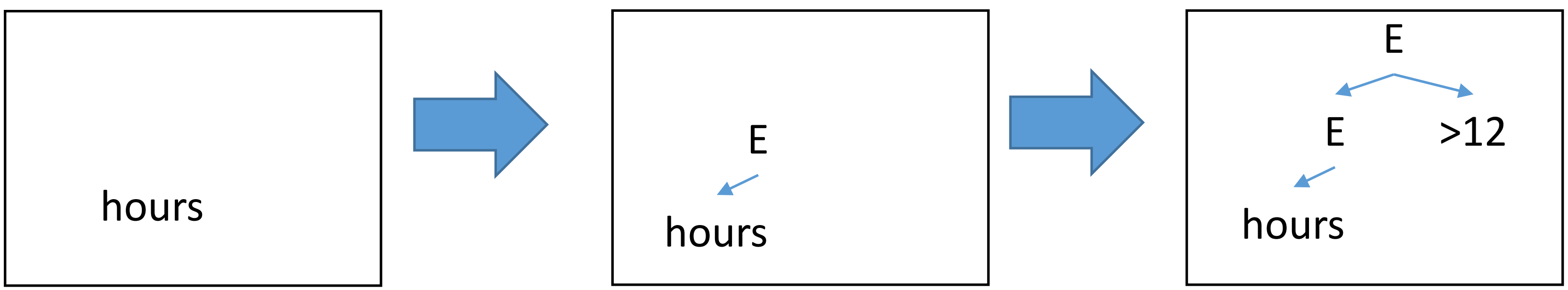
- 如果左下采用E->x的概率极大，而右下采用E->y的概率较低，则上面的顺序能显著减少搜索时间
- 需要根据应用特点定义非终结选择策略

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
- 通过采用合适的扩展规则，求解效率可提高一倍以上

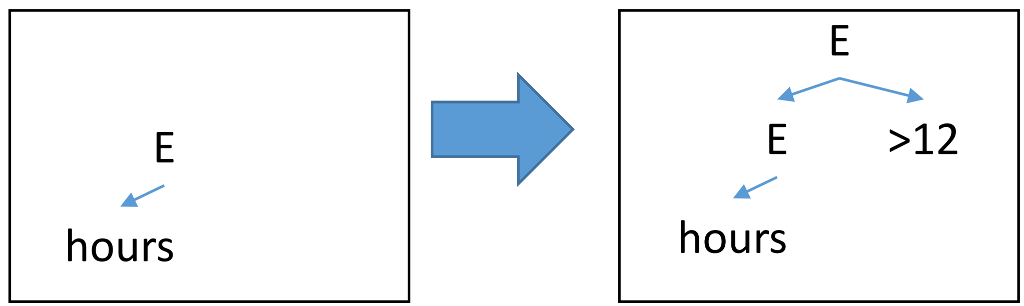
从上下文无关文法到扩展规则

$T \rightarrow E$

$E \rightarrow E \text{ “} > 12\text{”} \mid E \text{ “} > 0\text{”} \mid E \text{ “} + \text{”} E \mid \text{“}hours\text{”} \mid \text{“}value\text{”} \mid \ldots$

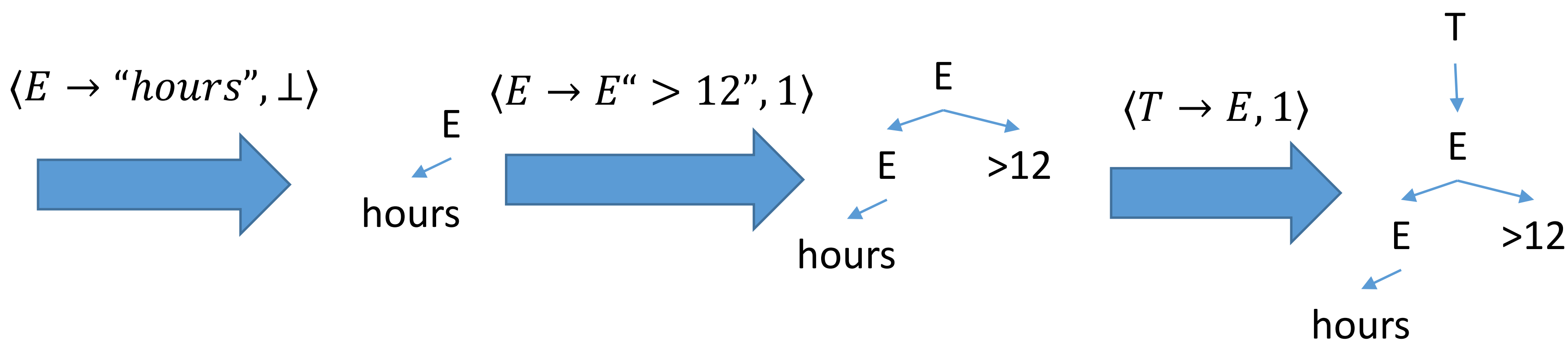
$\langle E \rightarrow \text{“}hours\text{”}, \quad \perp\rangle$
$\langle E \rightarrow \text{“}value\text{”}, \quad \perp\rangle$
$\langle E \rightarrow E \text{ “} > 12\text{”}, \quad 1\rangle$
$\langle E \rightarrow E \text{ “} + \text{”} E, \quad 1\rangle$
$\langle T \rightarrow E, \quad 1\rangle$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

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



自顶向下规则：$\langle E \rightarrow E\text{“} > 12\text{”}, 0\rangle$
如果根节点已经产生，产生整颗子树
创建规则：$\langle E \rightarrow \text{“}hours\text{”}, \perp\rangle$
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
$\begin{array}{c} (T \longrightarrow E, 1) \\ \uparrow \\ (E \longrightarrow E \text{ “} > 12\text{”}, 1) \\ \uparrow \\ (E \longrightarrow \text{“hours”}, \perp) \end{array}$	$\begin{array}{ccc} & (T \longrightarrow E, 1) & \\ & \uparrow & \\ & (E \longrightarrow E \text{ “} +\text{”} \ E, 1) & \\ \nearrow & & \searrow \\ (E \longrightarrow \text{“hours”}, \perp) & & (E \longrightarrow \text{“value”}, 0) \end{array}$

抽象语法树 -> 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

$$T \rightarrow E$$

$$E \rightarrow E \text{ “} > 12\text{”} \mid E \text{ “} > 0\text{”} \mid E \text{ “} + \text{”}\ E \mid \text{“}hours\text{”} \mid \text{“}value\text{”} \mid \ldots$$

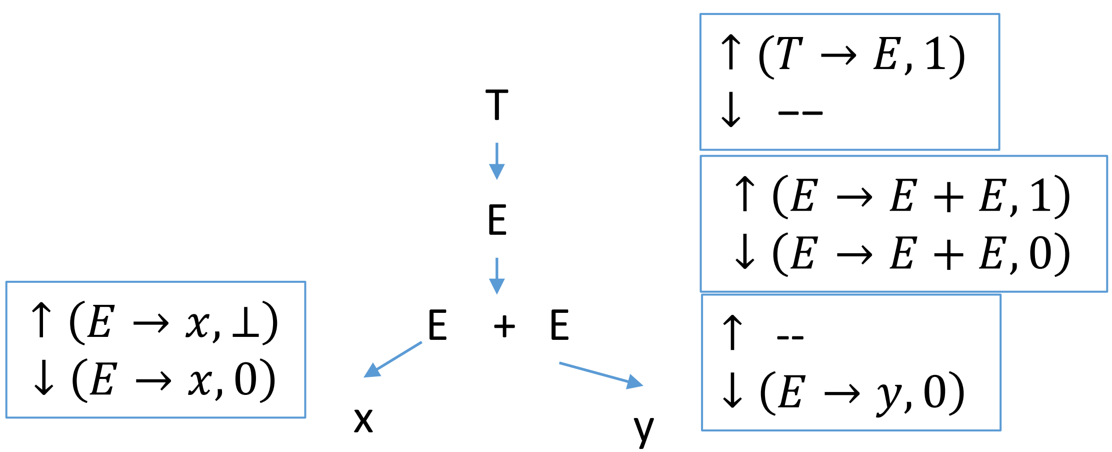
$$
\begin{array}{ll}
\langle E \rightarrow \text{“}hours\text{”}, & \perp\rangle \\
\langle E \rightarrow \text{“}value\text{”}, & \perp\rangle \\
\langle E \rightarrow E \text{ “} > 12\text{”}, & 1\rangle \\
\langle E \rightarrow E \text{ “} + \text{”}\ E, & 1\rangle \\
\langle T \rightarrow E, & 1\rangle \\
\langle E \rightarrow E \text{ “} > 12\text{”}, & 0\rangle \\
\langle E \rightarrow E \text{ “} + \text{”}\ E, & 0\rangle \\
\langle E \rightarrow \text{“}hours\text{”}, & 0\rangle \\
\langle E \rightarrow \text{“}value\text{”}, & 0\rangle
\end{array}
$$

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

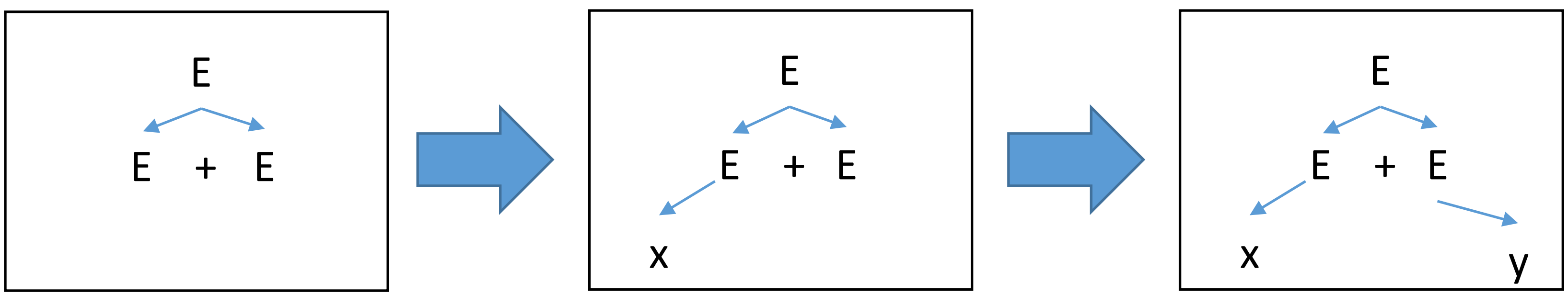
吉如一
北京大学博士生

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->[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



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

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|⊥]S->[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

<table>
<tr><th>带祖先和概率下界的VSA</th><th>概率</th><th>说明</th></tr>
<tr><td>[acc|⊥|0.3]S->[a|S|0.33]S+[cc|S|0.33]S</td><td>0.9</td><td>0.3/0.9=0.33</td></tr>
<tr><td>[cc|S|0.33]S-~~>[c|S|3.3]S~~ + ~~[c|S|3.3]S~~</td><td>0.1</td><td>0.33/0.1=3.3</td></tr>
<tr><td>| ~~y~~</td><td>0.3</td><td>0.33/0.3=1.1</td></tr>
</table>

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
⊥	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|⊥|0.3]S -> ~~[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

参考文献

- Yingfei Xiong, Bo Wang, Guirong Fu, Linfei Zang. Learning to Synthesize. GI'18: Genetic Improvment 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 Programing 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. AAAI'19: Thirty-Third AAAI 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. AAAI'20: Thirty-Fourth AAAI 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.