



软件分析

程序合成：其他方法

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

- Expr
→ **ite** Cond Expr Expr
| Term



Rajeev Alur
Upenn教授
STUN思想、Eusolver
和Euphony方法的提出者



吉如一
北京大学博士生
PolyGen提出人



基本流程(1/3)

- 首先枚举一组Cond和Term
 - 对于Term，记录所覆盖的example
 - 基于Cond，记录将example分成的两个组
- 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 terms	Cond	
				$x=3$	$x < y$
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Term集合
- 在这里选择x和y

x	y	ret	covering terms	Cond	
				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Cond构造最终程序
 - 选择一组Cond对example分类，让每个分类都能被一个所选Term覆盖
 - 类似于机器学习中的决策树构建算法
- 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 $\text{if}(x < y) \text{ then } x \text{ else } y$

x	y	ret	covering terms	Cond	
				x=3	x<y
1	2	2	y	false	true
3	3	3	x,y	true	false
5	2	5	x	false	false



控制程序的总大小

- 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
- 如何控制合成程序的总大小？
- Eusolver和Euphony
 - 控制Term和Cond的大小
 - 从小到大枚举，给出大小上限
 - 从比较小的上限开始，逐步放宽
 - 控制Term的个数
 - 采用尽量少的Term来覆盖
 - 控制Cond的个数
 - 采用决策树算法来得到尽量小的决策树



控制程序的总大小

- 问题：决策树算法为模糊分类设计，不适合精确的程序合成
 - 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采用更有保障的随机采样来控制程序的总大小

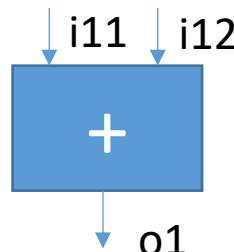


约束求解法



约束求解法

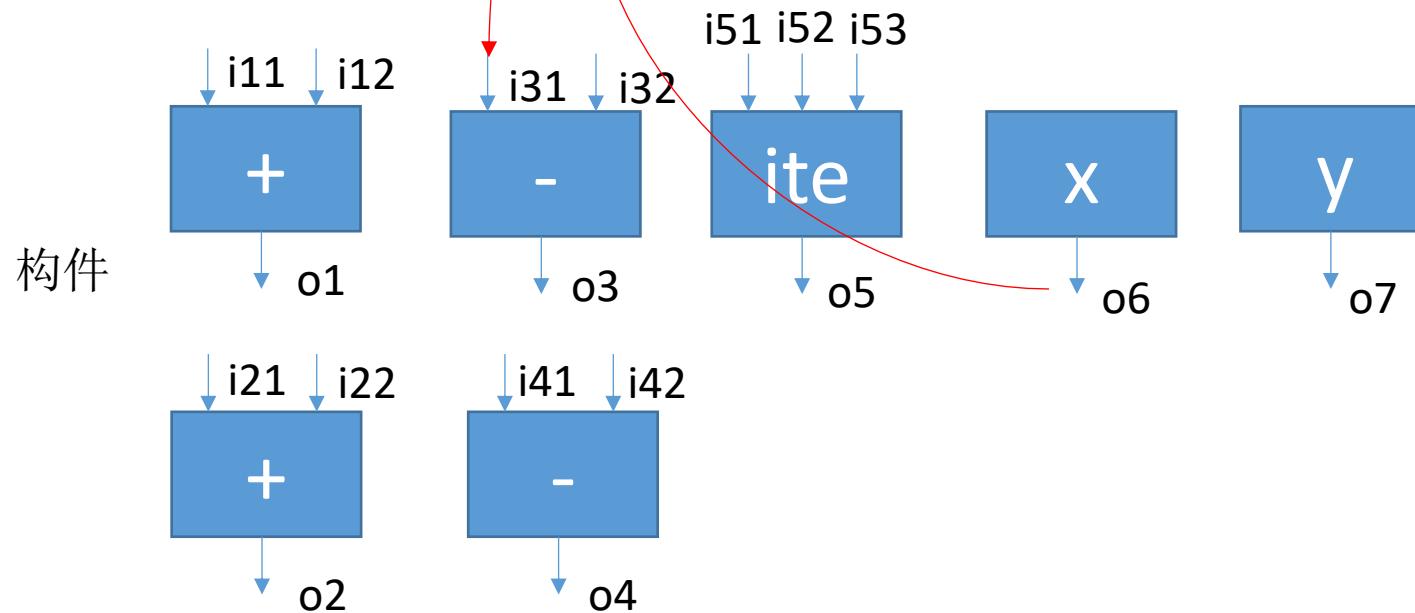
- 将程序合成问题整体转换成约束求解问题，由 SMT求解器求解
- 代表工作：基于构件的程序合成
Component-Based Program Synthesis
 - 获得ICSE十年最有影响力论文奖
- 将产生式看做构件，并且数量有限
 - 1号产生式： $Expr_{o1} \rightarrow Expr_{i11} + Expr_{i12}$



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}, \dots
- l_{o1}, l_{o2}, \dots
- l_o : 程序输出



产生约束

- 对于任意测试 t 产生约束，如：
 - $v_x = 1 \wedge v_y = 2 \rightarrow v_o = 2$
- 对所有构件产生语义约束：
 - $v_{o1} = v_{i11} + v_{i12}$
- 对所有的构件产生非终结符约束：
 - $N_{o1} = \text{"Expr"}$
- 对所有的输入输出标签对产生连接约束：
 - $l_{o1} = l_{i21} \rightarrow v_{o1} = v_{i21}$
 - $l_{o1} = l_{i21} \rightarrow N_{o1} = N_{i21}$
- 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)
 - $l_{i11} \geq 1$
- 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
 - $l_{i11} < l_{o1}$
- 结合CEGIS可以求解任意规约

额外变量：

- v_{o1} : 程序执行中 $o1$ 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 $l_{ox} \dots$, 直接用 l_{ixx} 的值表示应该连接第几号输出?



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：

```
Expr ::= x | y  
       | Expr + Expr  
       | (ite BoolExpr Expr Expr)  
BoolExpr ::= BoolExpr ∧ BoolExpr  
           | ¬BoolExpr  
           | Expr ≤ Expr
```

- 规约：

$$\forall x, y : \mathbb{Z}, \quad \max_2(x, y) \geq x \wedge \max_2(x, y) \geq y
 \wedge (\max_2(x, y) = x \vee \max_2(x, y) = y)$$

- 期望答案：ite ($x \leq y$) 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：
 - $[2]\text{Expr} \rightarrow [1]\text{Expr}+[1]\text{Expr}$
 - 对应的，称 $\text{Expr} \rightarrow \text{Expr}+\text{Expr}$ 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
- 删减从起始符号和期望输出不可达的产生式

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

$[2]\text{Expr} \rightarrow [2]x$

$[1]\text{Expr} \rightarrow [1]y$

$[3]\text{Expr} \rightarrow [2]\text{Expr} + [1]\text{Expr}$



从样例得到文法例子1

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



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：

$$S \rightarrow AC$$

$$A \rightarrow aAb \mid ab$$

$$C \rightarrow cC \mid c$$

$$S' \rightarrow A'C'$$

$$A' \rightarrow aA' \mid a$$

$$C' \rightarrow bC'c \mid bc$$

$S \cap S'$ 不是上下文无关语言

- 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

- 给每一个产生式赋唯一ID
 - $P_1: S \rightarrow S + S$
 - $P_2: S \rightarrow 1$
- 可以把产生式改写成如下形式
 - $S \rightarrow P_1(S, S)$
 - $S \rightarrow P_2()$
 - 其中 P_1, P_2, \dots 看做终结符
- 括号保证了字符串被解析为唯一的结构，即文法求交等价于语法树集合求交
- 实际不需要做这样的改写，只要产生语法树集合求交的效果就行



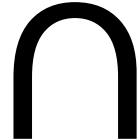
文法求交

- 给定原始产生式
 - $N_0 \rightarrow P(N_1, N_2, \dots)$
- 如果两个文法中分别存在两个带约束产生式
 - $[c_0]N \rightarrow P([c_1]N_1, [c_2]N_2, \dots)$
 - $[c'_0]N \rightarrow P([c'_1]N_1, [c'_2]N_2, \dots)$
- 求交得到
 - $[c_0, c'_0]N \rightarrow P([c_1, c'_1]N_1, [c_2, c'_2]N_2, \dots)$
- 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 -> ~~c~~
[cc, c]S -> ~~c~~
[ac, a]S -> ~~c~~
[c, ac]S -> ~~c~~
[ac, aa]S -> ~~[a, a]S+[c, a]S~~
[c, a]S -> ~~c~~



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
 - $[2]y, [1]\text{Expr}+[1]\text{Expr}, \text{if}([\text{true}]\text{BoolExpr}, [2]\text{Expr}, [*]\text{Expr}), \text{if}([\text{false}]\text{BoolExpr}, [*]\text{Expr}, [2]\text{Expr})$
 - 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



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
- $[2]\text{Expr} \rightarrow y | [1]\text{Expr} + [1]\text{Expr}$ |
 $\quad \text{if}([\text{true}]\text{BoolExpr})[2]\text{Expr} \quad [*]\text{Expr}$ |
 $\quad \text{if}([\text{false}]\text{BoolExpr})\dots$
- $[1]\text{Expr} \rightarrow x$
- $[*]\text{Expr} \rightarrow \dots$
- $[\text{true}]\text{BoolExpr} \rightarrow \text{true} | \neg [\text{false}]\text{BoolExpr}$ | $[2]\text{Expr} \leq [2]\text{Expr}$ | $[1]\text{Expr} \leq [2]\text{Expr}$ | $[1]\text{Expr} \leq [1]\text{Expr}$ | \dots



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 → 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
```



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.



基于概率的方法



很多应用需要概率大的程序

文心大模型3.5

你好！最大子段和问题是给定一个整数的列表，然后求这个列表中连续的一段，其中这一段的整数的和最大，求这个最大的和。你能写一个程序求解该问题吗？

当然可以，以下是一个Python程序，用于求解最大子段和问题：

```
python
1 def maxSubarraySum(nums):
2     if not nums:
3         return 0
4
5     max_sum = nums[0]
6     for i in range(1, len(nums)):
7         max_sum = max(nums[i], max_sum + nums[i])
8
9     return max_sum
10
11 # 示例用法
12 nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
13 print(maxSubarraySum(nums)) # 输出 15
```

在这个程序中，我们定义了一个名为 `maxSubarraySum` 的函数，它接受一个整数列表作为输入，并返回这个列表中连续的一段，其中这一段的整数的和最大。

在函数中，我们首先检查输入列表是否为空，如果是，则返回0。否则，我们初始化一个变量 `max_sum` 为列表中的第一个元素，然后遍历列表中的所有元素。对于每个元素，我们将其与 `max_sum` 进行比较，并更新 `max_sum` 的值。最后，我们返回 `max_sum` 作为函数的结果。

在示例用法中，我们定义了一个整数列表 `nums` 并调用 `maxSubarraySum` 函数求解最大子段和问题。最后，我



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
 - $prog = \operatorname{argmax}_{prog \in Prog \wedge prog \vdash spec} P(prog)$
- 给定不同的 P , 可以实现不同任务。
 - 如果 P 估计程序满足给定自然语言需求的概率, 那么可以完成从自然语言的代码生成
 - 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



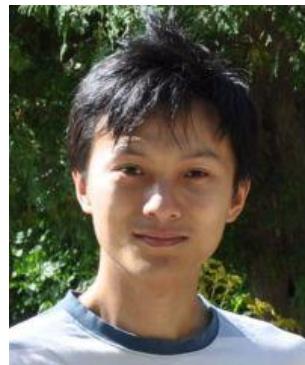
基本算法：穷举

- 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）



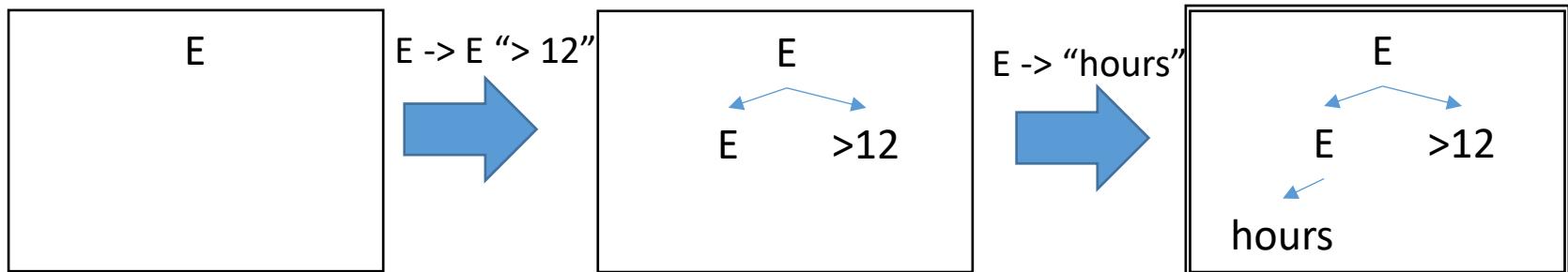
熊英飞
北京大学副教授



王博
北京交通大学讲师
北京大学博士



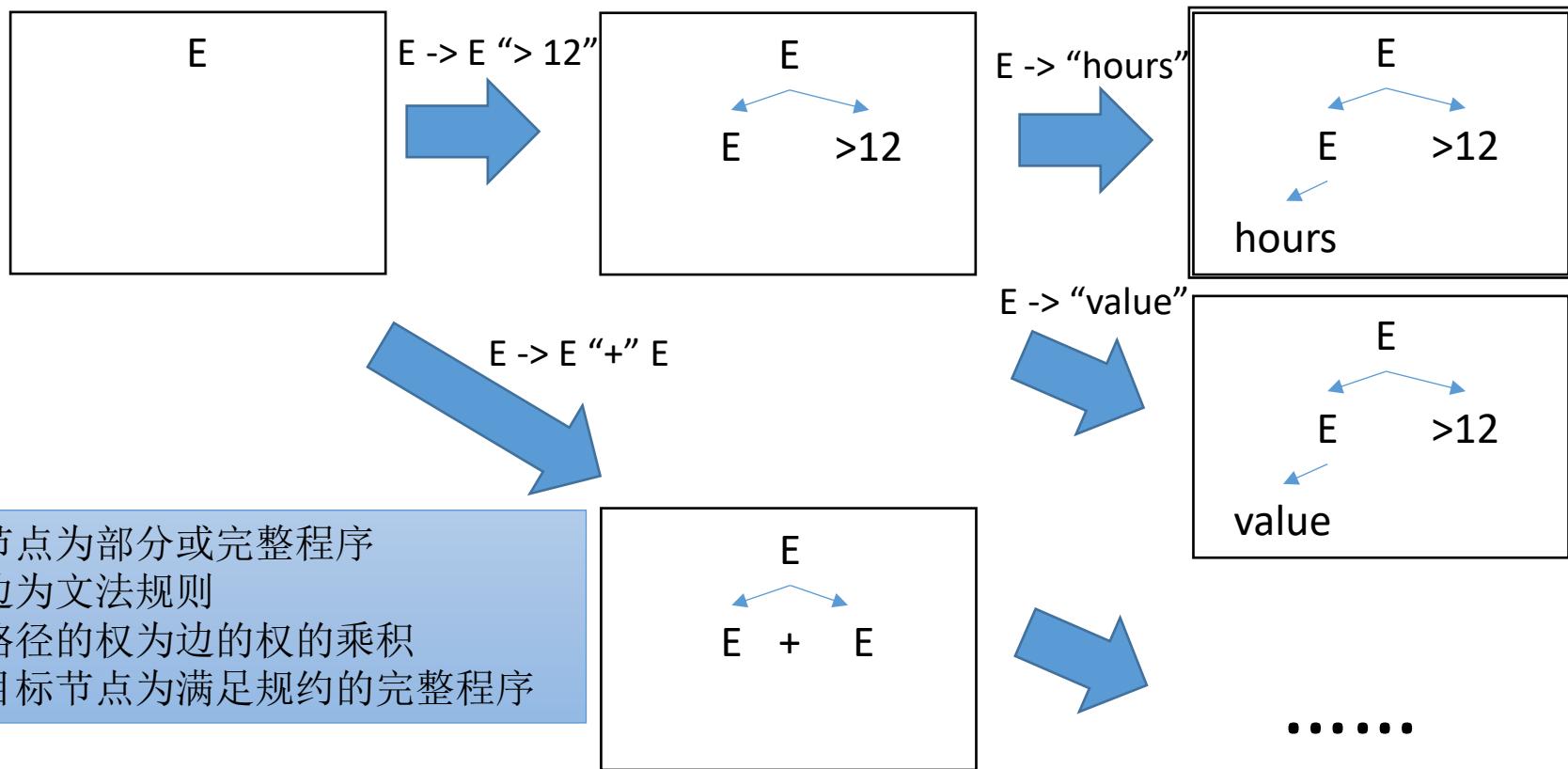
规则展开概率模型



- 按某种固定顺序选择节点展开
- $P(prog) = \prod_i P(rule_i | rule_1, \dots rule_{i-1})$
 - $rule_i$: 展开程序所用的文法规则
- 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





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
 - $\forall prog: \hat{P}(rule) \geq P(rule \mid prog)$
 - 在概率模型简单的时候可以遍历获得上界
 - 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



和大模型的关系

- 规则展开概率模型等价于将代码建模为文法规则编码的序列
 - 保证文法正确性
 - 生成过程保留文法结构，可以应用程序分析剪枝
- 大模型将代码和文本统一建模为Token序列
 - 不保证文法正确性
 - 生成过程不保留文法结构，Token序列甚至不和词法分析对应，程序分析需要先实现partial parser
- 大模型也能采用之前介绍的所有搜索算法，但由于计算资源消耗多，所以通常采用随机采样



和大模型结合

- 可以将程序表示成文法规则序列来训练大模型
- 在各个规模的实验均表明显著优于Token序列表示
- 原因：
 - 和语义对应得更好
 - 更容易解析



原因1：和语义更好的对应

- 相同语义

- ```
if (x<0) y=y+1;
```
- ```
if (x < 0) {  
    y = y + 1;  
}
```

在语法规则序列下相似
在单词序列下不同

- 不同语义

- ```
for i in range(1, 6):
 x = x + 1
 sum = sum + x
```
- ```
for i in range(1, 6):  
    x = x + 1  
    sum = sum + x
```

在语法规则序列下不同
在单词序列下相似



原因2：更容易解析

- 编程语言的解析越容易，模型表现就越好。
- 在更大模型（1-1.5B），Python语言的更多表示上的结果一致。

If n is an integer and
101 * n² ≤ 3600,
what is the greatest
possible value of n?
n0 = 101.0 n1 = 2.0
n2 = 3600.0

a. Question

1. import math
2. n0 = 101.0
3. n1 = 2.0
4. n2 = 3600.0
5. t0 = n2 / n0
6. t1 = math.sqrt(max(0, t0))
7. answer = math.floor(t1)

b. Answer in DSL_{LR(1)}

1. import math
2. <exp> n0 = 101.0
3. <exp> n1 = 2.0
4. <exp> n2 = 3600.0
5. <exp> t0 = <exp> / n2 n0
6. <exp> t1 = <exp> <call> <exp> <attr> math.sqrt(<exp> <call> max(0, t0))
7. <exp> answer = <exp> <call> <exp> <attr> math.floor(t1)

d. Answer in DSL_{LL(2)}

1. import math
2. n0 = 101.0
3. n1 = 2.0
4. n2 = 3600.0
5. t0 = / n2 n0
6. t1 = <call> <attr> math.sqrt(<call> max(0, t0))
7. answer = <call> <attr> math.floor(t1)

c. Answer in DSL_{LL(1)}

1. import math
2. n0 = 101.0 ; n0
3. n1 = 2.0 ; n1
4. n2 = 3600.0 ; n2
5. t0 = n2 / n0 ; t0
6. t1 = math.sqrt(max(0, t0)) ; t1
7. answer = math.floor(t1) ; ans

e. Answer in DSL_{CSG}

Language	Mean(%)
DSL _{LL(1)}	82.00
DSL _{LL(2)}	81.74
DSL _{LR(1)}	81.14
DSL _{NCFG}	80.41



原因2：更容易解析

- 语法序列表示属于LL(1)
- 比大多数主流编程语言都要容易解析
 - Python：非上下文无关语言
 - Java：LR



作业

1. 如果当前只有一个输入输出样例，考虑自底向上合成，基于空间表示的方法相对可观察等价在效率上是否存在优势？如果存在，请给出一个例子；如果不存在，请说明理由。
2. 如果当前只有一个输入输出样例，对比自底向上的空间表示合成和合一化程序合成，语法如下：
 - $E \rightarrow \text{ite } C \ T \ T$
 - $C \rightarrow x > 1 \mid x > y \mid y < 1 \mid \dots$
 - $T \rightarrow x \mid y \mid x + y \mid \dots$

其中C和T的所有产生式的右边都不再有非终结符。在这个场景中，合一化程序合成相比空间表示的程序合成在效率上是否存在优势？如果存在，请给出一个例子；如果不存在，请说明理由。



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
- 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.