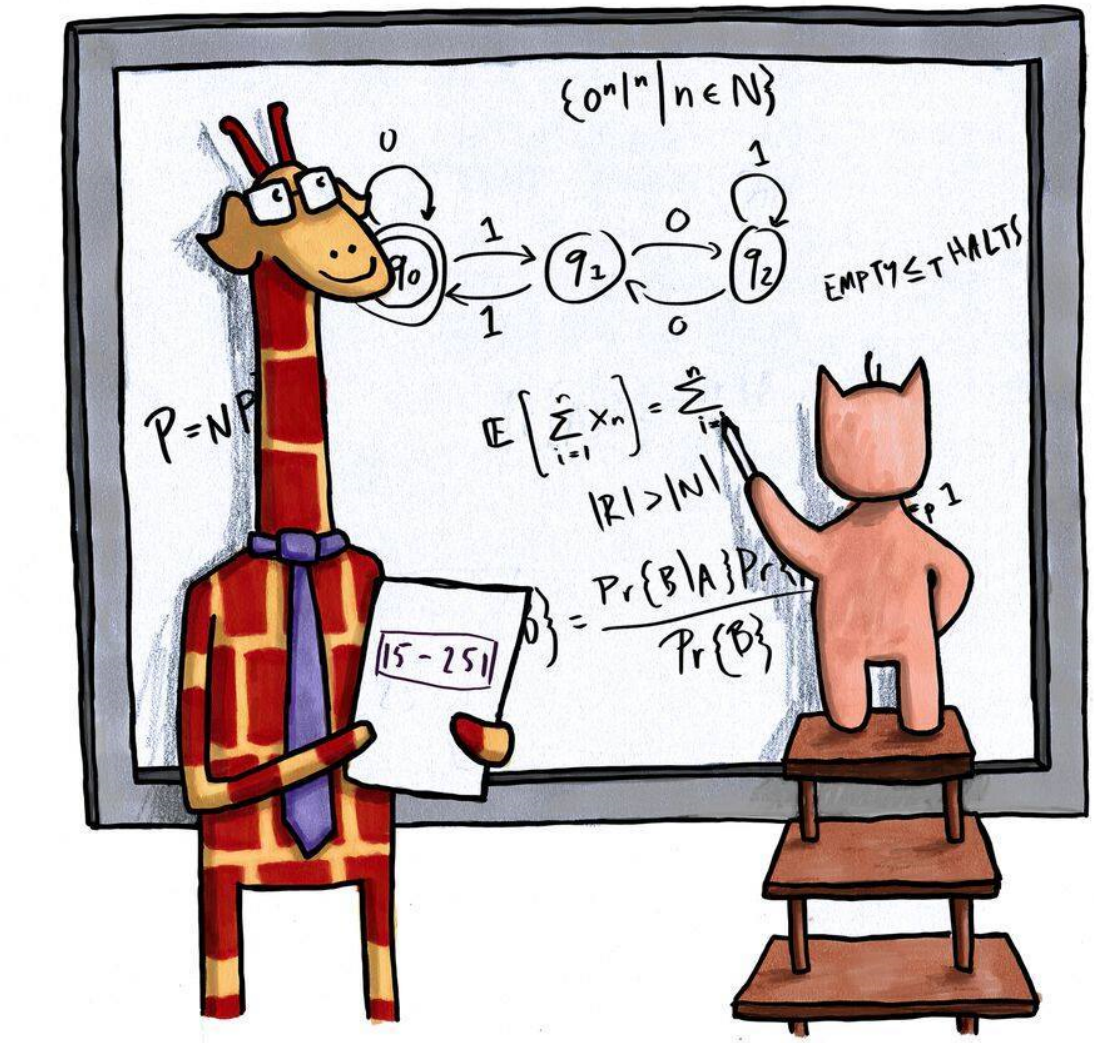
程序综合报告

唐雯豪, 易普, 谢睿峰

EECS, PKU
January 6, 2021



Outline

1 概述

2 方法一：VSA

3 方法二：直接法

4 方法三：DSL 增强的搜索

5 方法四：基于 Rosette 的暴力展开

6 总结

概述

我们小组设计并实现了四种方法，最后提交的程序使用了其中三种方法并行求解，可以解出 open tests 中所有测试用例，并且具有一定泛化能力。

Outline

1 概述

2 方法一：VSA

3 方法二：直接法

4 方法三：DSL 增强的搜索

5 方法四：基于 Rosette 的暴力展开

6 总结

算法过程

和上课讲的基本相同：

1. 随机生成几个样例
2. 从样例中生成 VSA 并求交
3. 从求交后的 VSA 中生成任意一个程序
4. 扔给 SMT solver，有反例的话从反例生成 VSA 并跟当前 VSA 求交，重复上一步；否则得到结果

效果

写得太差了，只能解出 three 和 max2
max3 中随着样例数量增加，VSA 中非终结符数量的增长：33 $\rightarrow$ 208 $\rightarrow$ 1466 $\rightarrow$ 10251 $\rightarrow$ 71990 $\rightarrow$ VSA 求交函数爆栈
沉没成本，果断放弃

Outline

1 概述

2 方法一：VSA

3 方法二：直接法

4 方法三：DSL 增强的搜索

5 方法四：基于 Rosette 的暴力展开

6 总结

观察

1. SyGuS 问题直接给了我们对函数结果的约束
2. 大多数对函数结果的约束都是等号约束
    - 对于 open tests 来说，除了 max 系列问题之外都是只有等号约束
    - 并且除了 three 问题之外，所有等号约束都是函数结果只出现在等号一边（也就是没有递归约束）

$\rightarrow$ **直接**从约束中综合出程序。

算法过程

主要想法：提取每个对函数结果的等号约束成立时的条件，然后把整个程序写成一堆 if 语句。
具体分为三个步骤：

1. 约束规范化
2. 分支提取
3. 局部搜索

约束规范化

1. 将所有 constraint 语句用 and 连起来
2. 对于传递了常数的函数调用，将常数参数作为一个条件前置，比如将 $f\,3\,\ldots = y$ 转化成 $(x = 3) \to f\,x\,\ldots = y$
3. 将 $A \to B$ 转化成 $\neg A \wedge B$
4. 使用 $\neg(A \wedge B) = \neg A \vee \neg B,\ \neg(A \vee B) = \neg A \wedge \neg B,\ \neg\neg A = A$ 将 $\neg$ 下传并消减。
5. 将连续的 and 和 or 合并成一个，比如将 $(and\ x\ (and\ y\ z))$ 转化成 $(and\ x\ y\ z)$

最终的约束树中的只有多分支的 and 和 or，所有 not 都在最底层。

分值提取

从规范化后的约束中提取每个对函数结果的等号约束成立时的条件。
在这里我们假设约束确实对应了一个函数（可以是 partial 的）。
只需要分别考虑遇到 and 和 or 时应该怎么办：

- **and**: 对于所有有函数结果约束的分支 x，将其所有没有函数结果约束的兄弟分支的 and 作为 x 中约束成立的条件的一部分。
- **or**: 对于所有有函数结果约束的分支 x，考虑其所有没有函数结果约束的兄弟分支的 and 取反后作为 x 中约束成立的条件的一部分。

~~正确性尚未证明~~。

局部搜索

问题：SyGuS 问题中给的语法没有涵盖约束中出现的所有语法。

1. 有的语法没有给出，比如没有 and
2. 有的常数没有给出，比如 5 或者 False
3. 没有不动点，写不了递归

问题一和问题二都可以通过对于语法、常数或者含有它们的局部进行搜索的方法解决，也可以内置一些语法糖。搜索或者内置的结果举例：

- $and(x, y) = ite(x, y, False)$
- $False = x < x$
- $x < 5 = x + x < 10$

问题三没法解决，交给其它方法做

一个简单的例子

```
(set-logic LIA)
(synth-fun findIdx ( (y1 Int) (y2 Int) (k1 Int)) Int
  ((Start Int ( 0 1 2 y1 y2 k1 (ite BoolExpr Start Start)))
   (BoolExpr Bool ((< Start Start) ( ≤ Start Start) (> Start Start) ( ≥ Start Start)))
  )
)
(declare-var x1 Int)
(declare-var x2 Int)
(declare-var k Int)
(constraint (⇒ (< x1 x2) (⇒ (< k x1) (= (findIdx x1 x2 k) 0))))
(constraint (⇒ (< x1 x2) (⇒ (> k x2) (= (findIdx x1 x2 k) 2))))
(constraint (⇒ (< x1 x2) (⇒ (and (> k x1) (< k x2)) (= (findIdx x1 x2 k) 1))))
(check-synth)
```

```
['and',
 ['or',
  ['not', ['<', 'x1', 'x2']],
  ['not', ['<', 'k', 'x1']],
  ['=', ['findIdx', 'x1', 'x2', 'k'], ('Int', 0)]],
 ['or',
  ['not', ['<', 'x1', 'x2']],
  ['not', ['>', 'k', 'x2']],
  ['=', ['findIdx', 'x1', 'x2', 'k'], ('Int', 2)]],
 ['or',
  ['not', ['<', 'x1', 'x2']],
  ['not', ['>', 'k', 'x1']],
  ['not', ['<', 'k', 'x2']],
  ['=', ['findIdx', 'x1', 'x2', 'k'], ('Int', 1)]]]
```

```
['ite',
 ['ite', ['<', 'y1', 'y2'], ['<', 'k1', 'y1'], ['<', 'y1', 'y1']],
 0,
 ['ite',
  ['ite', ['<', 'y1', 'y2'], ['>', 'k1', 'y2'], ['<', 'y1', 'y1']],
  2,
  1]]
```

效果

- open tests 中除了 max 和 three 都能解，并且可以在 1s 内解完。
- 理论上对于所有 SyGuS 问题，只要它对函数结果的约束只有等号约束，并且函数结果只在等号一边出现，都可以用该方法解出。
- ~~并没有什么实用性。~~
- ~~或许可以推广到部分带有非等号约束的问题。~~

Outline

1 概述

2 方法一：VSA

3 方法二：直接法

4 方法三：DSL 增强的搜索

5 方法四：基于 Rosette 的暴力展开

6 总结

动机

1. “目前 Synthesis 方法的效率高度依赖 DSL 的设计”
2. MaxFlash 论文里和 TText 对比时说 TText 的 DSL 有 regex 所以它解的更快。
3. 加上 $max(x, y) = ite(x < y, y, x)$ 这个语法糖后 max 问题的程序可以随便写。

$\rightarrow$ 在 DSL 中加入一些更强的构造，并优先使用这些构造进行搜索，搜出程序后再加一个解语法糖的过程。

效果

max 问题都可以很快的搜出来。
我们组提交的测试用例是 min15，也可以很快的搜出来。
但我们现在只加了 max 和 min 这两个语法，只对含有需要他们的程序有优化效果。

Outline

1 概述

2 方法一：VSA

3 方法二：直接法

4 方法三：DSL 增强的搜索

5 方法四：基于 Rosette 的暴力展开

6 总结

算法过程

- Rosette 是 Racket 语言的一个扩展，提供了 solver-aided programming 的能力，简单来说就是可以直接把代码扔给 SMT solver。
- 并且 Rosette 提供了一个 $define-synthax$ 语法构造，可以定义一个 CFG 并且暴力展开 k 层，然后扔给 SMT solver 求解。
- 于是我们在外面套了一个对于 k 的迭代加深然后暴力展开求解。
- 实际上这个东西非常慢，不对 SyGuS 给出的语法做一些剪枝的话，max3 都要解几分钟。

Outline

1 概述

2 方法一：VSA

3 方法二：直接法

4 方法三：DSL 增强的搜索

5 方法四：基于 Rosette 的暴力展开

6 总结

总结

- 我们最后的方案是并行跑直接法、DSL 增强的搜索和基于 Rosette 的暴力展开。
- 实际上只需要前两种方法就可以解出 open tests 中的所有问题。（还带着 Rosette 的原因是因为时间仓促搜索法有个搜不出 three 的 bug 还没来得及修）
- 实现上，使用了 Python, Haskell, Racket 三种语言，遇到了一系列~~徒手造轮子~~和并行问题。

谢谢聆听.