# Genetic Synthesis: Finding the Best Specification for Synthesis Problems

2021-2 Advanced Software Analysis

Term Project

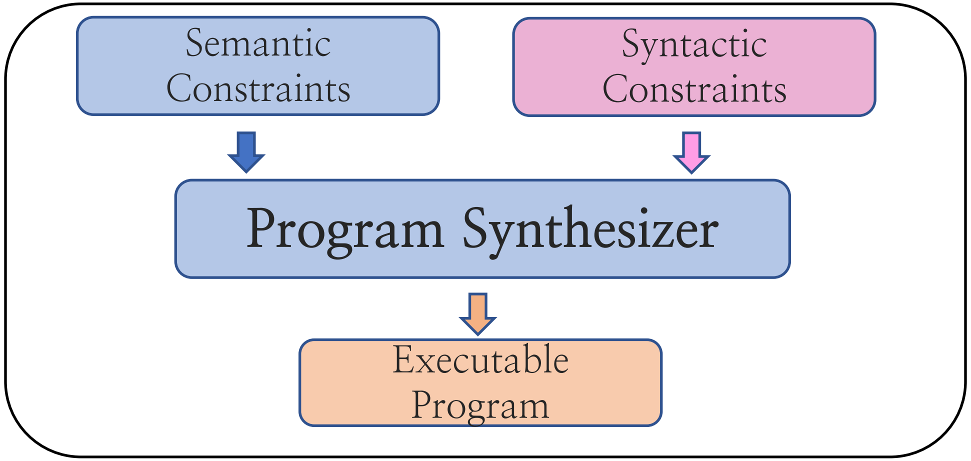
김요엘

kimyoel2305@gmail.com

# Background: What is Program Synthesis?

- The task of discovering an **executable program** from **user intent** expressed in the form of some **constraints.**

Syntax-Guided Synthesis(SyGuS)*




*Alur, Rajeev, et al. *Syntax-guided synthesis*. IEEE, 2013.

# The details of Program Synthesis

The details of syntactic constraints:

- restriction of available parameters, constants, and operators (program search space).

The details of semantic constraints:

- whether candidate solutions selected in the search space satisfy given I/O examples.

**specification**

| Syntax | | I/O examples |
|---|---|---|
| $x, 2, 3$ | | $f(0) = 3$ |
| $(* \; Int \; Int)$ | + | $f(1) = 5$ |
| $(+ Int \; Int)$ | | $f(2) = 7$ |

→ EuSolver* →

solution

$f(x) = 2x + 3$

*Alur, Rajeev, Arjun Radhakrishna, and Abhishek Udupa. "Scaling enumerative program synthesis via divide and conquer." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 2017.

# The details of Program Synthesis

The details of syntactic constraints:

- restriction of availabl … ce).

My goal is to find this kind of specification that helps the synthesizer to discover the best solution.

The details of semanti

- whether candidate s … les.

**specification**

Syntax
$x, 2, 3$
$(* \; Int \; Int)$
$(+ Int \; Int)$

+

I/O examples
$f(0) = 3$
$f(1) = 5$
$f(2) = 7$

EuSolver*

solution
$f(x) = 2x + 3$

*Alur, Rajeev, Arjun Radhakrishna, and Abhishek Udupa. "Scaling enumerative program synthesis via divide and conquer." *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Berlin, Heidelberg, 2017.

# Why is it important?

- Verification of a software containing external libraries **without source codes.**

- Static code verifiers such as symbolic execution, cannot explore these externals.

- Common alternatives: returning symbolic, non-deterministic, or random values.
- However, they could generate a lot of **false alarms** because of low accuracy.

- Note that we can get I/O examples by logging externals.

- **Through program synthesis, we can synthesize externals more accurately than common alternatives.**

# Is it Really a Combinatorial Optimization?

- Suppose a target function $f$ with $k$ parameters, which uses integer/Boolean operators and constants, and returns an integer value.

- **Combination of parameters**: By binomial theorem, $\sum_{r=0}\binom{k}{r} = 2^k$.
- **Combination of operators**: $\{+, -, \times, \div, \%, ite, or, and, not, <, \leq, ==, >, \geq\}$, which means $\sum_{r=0}\binom{14}{r} = 2^{14}$.
- **Combination of constants**: −2,147,483,648 ~ 2,147,483,647, which means $\sum_{r=0}\binom{2^{32}}{r} = 2^{2^{32}}$.

- Theoretically, the number of syntactic constructions is $2^{k+14+2^{32}}$.

# Is it Really a Combinatorial Optimization?

- We got $n$ I/O examples after running $f$ $n$ times.
- Combination of I/O examples: $\sum_{r=0} \binom{n}{r} = 2^n$.
- In total, the number of specifications of $f$ is $2^{n+k+14+2^{32}}$.

# Design Decisions

- The size of initial population: **(the number of available processors) * 2.**
- Timeout for synthesizing each program (individual): **20 sec** (in the experiment).
- The number of I/O examples in the initial population: **10.**
- The least syntax includes '**1**', '**ite** (if-then-else)', '**==**', and **all parameters.**
- The range of available constants in the initial population: **−1 ~ 10.**

# Design Decisions

- Fitness function: **the accuracy of a synthesized program.** This is evaluated by executing 1,000 testcases.

- Stopping criteria: It stops if the **100% accuracy is achieved**, or total execution timeout (within 15 min) is exceeded.

- Selection operator: **tournament selection** with k = 5.

- Generational selection strategy: all individuals are sorted in the order of high scores in the fitness function, and only the number of the initial population is selected as the next population.

# Design Decisions (Crossover)

**spec1**

| Syntax | I/O examples |
|---|---|
| $x, y, z, 0, 1$ | $f(1,2,3) = 9$ |
| $(+\ Int\ Int)$ | $f(2,2,4) = 11$ |
| $(/\ Int\ Int)$ | $f(5,2,0) = 10$ |
| $(ite\ Bool\ Int\ Int)$ | |
| $(==\ \ Int\ Int)$ | |

+

**spec2**

| Syntax | I/O examples |
|---|---|
| $x, y, z, 1, 3$ | $f(1,2,3) = 9$ |
| $(*\ Int\ Int)$ | $f(2,2,2) = 9$ |
| $(ite\ Bool\ Int\ Int)$ | |
| $(==\ \ Int\ Int)$ | |
| $(or\ Bool\ Bool)$ | |

**new spec**

| Syntax | I/O examples |
|---|---|
| $x, y, z, 1, \mathbf{3}, \mathbf{9}$ | $f(1,2,3) = 9$ |
| $(+\boldsymbol{Int\ Int})$ | $f(2,2,4) = 11$ |
| $(ite\ Bool\ Int\ Int)$ | $f(5,2,0) = 10$ |
| $(==\ \ Int\ Int)$ | $f(2,2,2) = 9$ |

| | Synthesis result | Constant folding |
|---|---|---|
| **Spec1** | x+y+z+1+1+1 | x+y+z+3 |
| **Spec2** | 3*3 | 9 |

# Design Decisions (Mutation)

- adding in/output examples:

```
selected = random.choices(logs, rand(size/2));
examples.addAll(selected);
```

- deleting in/output examples:

```
selected = random.choices(examples, rand(size/2));
if(size > 10) examples.removeAll(selected);
```

- adding grammars:

```
for(operator : operators) if(25%) grammars.add(operator);
if(12.5%) grammars.add(rand(100)-1);
```

- deleting grammars:

```
for(grammar : grammars) if(12.5%) grammars.remove(grammar);
```

# Limitations and Future Works

➢ **Limitations:**

1. In some cases, it did not escape the **local maximum.**
2. The accuracy is affected by the number of I/O examples.
3. It cannot exceed the maximum performance of the synthesizer.

➢ **Future works:**

- We need to overcome the maximum performance of the synthesizer.
- ex) input space partitioning..