Learning to Synthesize

Yingfei Xiong, **Bo Wang**, Guirong Fu, Linfei Zang

Peking University

June 2, 2018

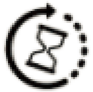
高可信软件技术
教育部重点实验室
Key Lab of High Confidence Software Technologies
(Peking University), Ministry of Education

PEKING UNIVERSITY
1898

Outline

- **GI and L2S**
- Components of L2S
- Application
- Conclusion

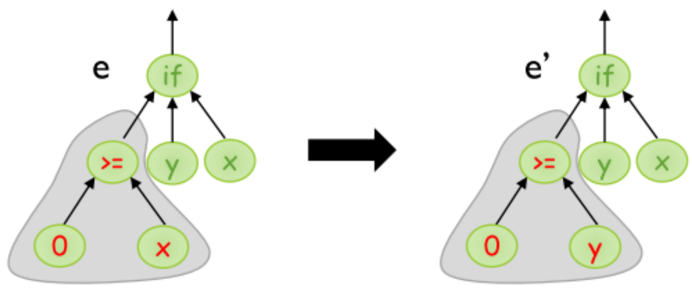
Genetic Improvement

- GI can systematically search for a program to fit some specifications in a large space
- Sometimes, we have strong and clear specifications
- Performance improvement:
  - Energy 
  - Execution time 

- However…

Genetic Improvement

- Many problems do not have strong specifications, or even have no specifications:
- Fit tests: generate-and-validate program repair
  - **GenProg**
  - **AE**



Genetic Improvement

- We aim to find the program fragments that are most-likely to be written under the current context.

```
public static long factorial(final int n){
    if( ... ){   }
}
```

```
...
Math.abs(n) < 1
n == Integer.Max_VALUE
n < 19
n < 21
...
```

- We define this problem as **program estimation**:
    - Given a context c, find program $s = \text{argmax}_s\, P(s \mid c)$?
- A sub-problem of program synthesis

Program Estimation

- Applications:
  - Context = input/output examples

Input	Output
0d 5h 26m	5:00
0d 4h 57m	4:30
0d 4h 27m	4:00
0d 3h 57m	3:30

  - Learning by examples

Program Estimation

- Applications:
  - Context = partial code

```
public static long fibonacci(int n) {
    if ( ?? ) return n;
    else return fibonacci(n-1) + fibonacci(n-2);
}
```

  - Code completion

Program Estimation

- Applications:
  - Context = natural language description

```
/**
 * Internal helper method for natural logarithm function.
 * @param x original argument of the natural logarithm function
 * @param hiPrec extra bits of precision on output (To Be Confirmed)
 * @return log(x)
 */
```

  - Code generation from natural language description

Program Estimation

- Applications:
  - Context = buggy program & at least one failed test

Passing Test

Failed Test

Buggy code

```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;
}
```

  - Test-based program repair

Challenges

- How to estimate the conditional $P(Prog \mid Context)$?
  - Should be consistent with other constraints, e.g., $P(invalid \mid Context) = 0$

```
Math.abs(n) < 1
n == Integer.Max_VALUE
```

```
n.length > 1
n == null
```

- How to find program s such that $P(\, s \mid context\,)$ is the largest?
  - The space of program is huge

Learning to synthesis (L2S)

- A general framework to address program estimation

- Combining four tools
  - **Rewriting rules**: defining a search problem
  - **Constraints**: pruning off invalid choices in each step
  - **Machine-learned models**: estimating the probabilities of choices in each step
  - **Search algorithms**: solving the search problem

Example – estimating conditions

- Condition bugs are common, **43% bugs in Defects4J** [1] are related to condition code blocks.
- Existing work can pinpoint incorrect condition
  - ACS
  - Nopol
- Can we generate a correct condition to fix the bugs?

```
  hours = convert(value);
+ if (hours > 12)
+   throw new ArithmeticException();
```

Missing condition

```
- if (hours >= 24)
+ if (hours > 24)
     withinOneDay=true;
```

Error condition

[1] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. 2018. **Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J**. CoRR abs/1801.06393 (2018).

Outline

- GI and L2S
- **Components of L2S**
- Application
- Conclusion

Rewriting Rules – estimating conditions

- Grammar:

$$
\begin{aligned}
E &\rightarrow V\ \text{“}\boxed{> 12}\text{”} \mid V\ \text{“}\boxed{> 0}\text{”} \mid V\ \text{“}\boxed{+}\text{”}\ V\ \boxed{\text{“} > 0\text{”}} \mid \ldots \\
V &\rightarrow \text{“hours”} \mid \text{“value”} \mid \ldots
\end{aligned}
$$

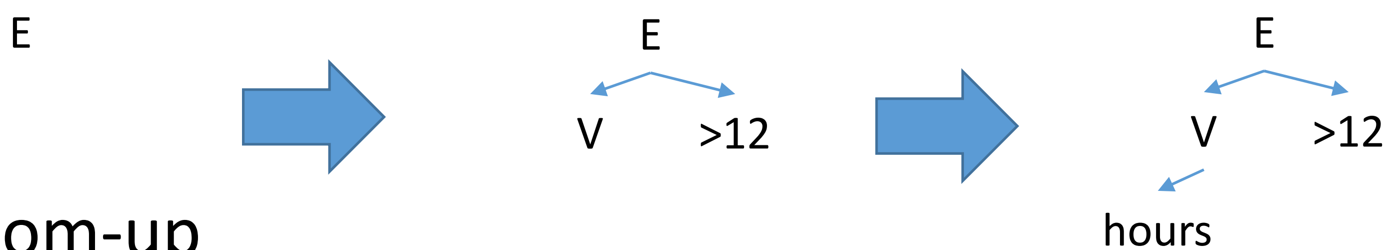
- Search Order:

(a) *hour* > 12 (b) $c[d] == null$

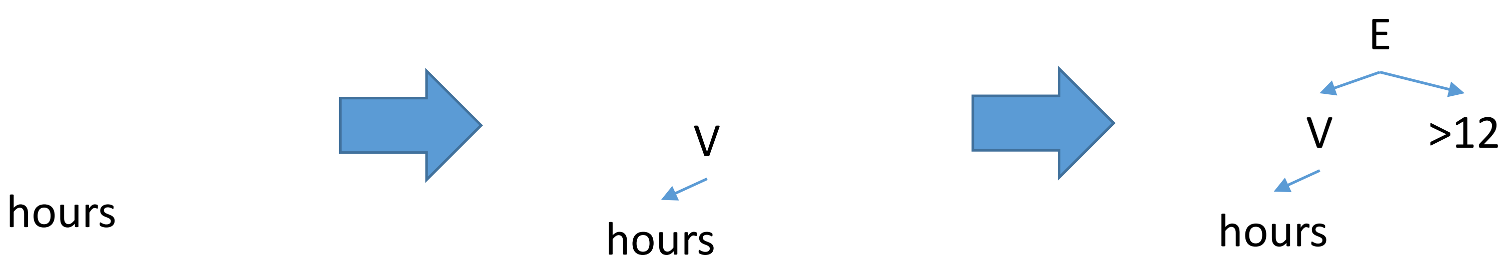
- The rewriting rules define the search space of conditional expressions

Rewriting Rules - Search Order

- A program may be completed in different orders
  - hours > 12
- Top-down



- Bottom-up



The order may greatly affect the performance of L2S.

Constrains

- Type constrains

```
if(E) E->boolean
V.equals("hello") V->String
V > 12 V->integer
```

- Dynamic value constrains

```
Passing Test 1
hour > 0

Passing Test 2
hour < 12
```

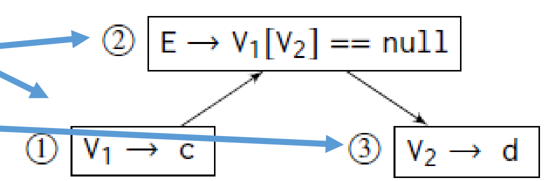
```
hour < 0
```

- Size constrains

```
V1 + V2 + V3 + V4 > V5 + V6 + V7 + V8
```

Machine-learned models

- The user chooses a machine learning method for each symbol E and V.
- The model is trained on a corpus of programs and their contexts, by a certain order.
  - A search process is re-enacted on each program and the choice of rules are stored as the training set
- Probability of a condition in bottom-up
  - P(Cond|Context) = P(V1|Context)
    * P(E| V1, Context)
    * P(V2| E, V1, Context)
    * …



(b) $c[d] == null$

Local Optimal ≠ Global Optimal

E_0

E → E " > 12"	0.3
E → E " > 0"	0.6

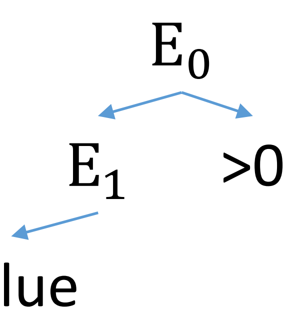
E_1

E → "hours"	0.1
E → "value"	0.2
E → E " + " E	0.05

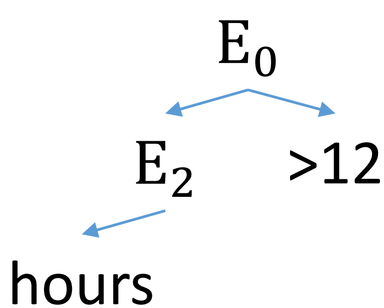
E_2

E → "hours"	0.8
E → "value"	0.1
E → E " + " E	0.05

0.6 * 0.2
= 0.12



0.3 * 0.8
= 0.24



Search Algorithm

- Beam search – a greedy method to solve the search problem
  - Top 200 for each E
  - Top 5 for each V
  - Discard P(E) < 0.01%

```
E -> V > 12                       0.3
E -> V > 24                       0.3
E -> V < 10000                    0.1
E -> V == Integer.Max_VALUE       0.1
E -> V * V == 0                   0.05
...
E -> Math.abs(V) < 1E-10          0.0001
E -> Math.floor(V) < V            0.0001
...
```

Discard Low Probability

- Other search algorithms may also be used

Outline

- GI and L2S
- Components of L2S
- **Application**
- Conclusion

Application - rewriting rule

- Grammar for conditional expressions
- Expr ->
  Math.abs($.value($)) < 1E-6

  $ > $

  $ < 0

  $ > $[1]

  …
- Var -> all available variables in the context
  - Parameters
  - Local variables
  - Non-constant fields (For non-static methods)

Application - rewriting rule

- Order : bottom up > top down

1: predict leftmost variable

entry
stop
n
maxIterations
overflow
value
epsilon
....

2: predict expr for each V1

$ == 0
Precision.compareTo($, 0d, $) > 0)
Double.isNan($)
...

$ > 0
$ == 1
$ < 0
...

...

3: predict remain variables

epsilon
overflow
maxIterations
n
value
....

2 Precision.compareTo($, 0d, $) > 0)

1 entry

3 epsilon

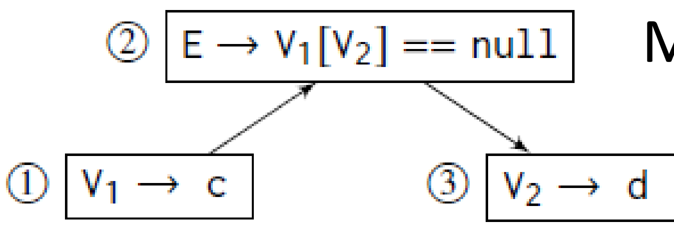
Precision.compareTo(entry, 0d, epsilon) > 0

Application - constrains

- Type constrains
  - Java grammar
- Size constrains
  - |V| <= 4
- Dynamic value constrains
  - Must pass all the tests

Application - machine-learned models

- Train a model for each step



(b) $c[d] == null$

Model 2: For Predicate

Model 1: For V1

Model 3: For the remaining variables

Application - machine-learned models

- Context Features
  - class information
    - class name, package, inherit relationship…
  - method information
    - name, type signature, static, constructor, size, comments…
  - code structure information
    - available variables
    - control flow points nearby: for, if, return, throw…
    - conditions nearby

```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) {
    if( ?? ){
        return a;
    }
    return b;
}
```

Application - machine-learned models

- Variable Features
  - name – encoded by textual similarity
  - Whether it is a meaningless name
  - type - integer, float, array, collection, string-related, others
  - modifier - static, final, volatile…
  - occurring time in other conditions

len
xLen
length
context
ctx

Meaningless identifiers:
a, b, l, j, p, q, u, v, x, y…

entry == 0
Double.isNaN(entry)
entry == 0

3 times

Application - machine-learned models

- Expression features
  - how many variables it will takes
  - expected types
  - occurred method name, operator and constant values
- Position feature
  - which variable positon to be expanded

Precision.compareTo($, 0d, $) > 0)

take 2 var

expect a double

Application - machine-learned models

- Learning algorithm - XGBoost
    - Tree-based models can easily handle unbalanced data
    - Training is fast (usually < 30 min)
- Why not deep learning?
    - the dataset is not large enough (the scale is $10^3 \sim 10^5$)
    - need complex preprocess for data
    - training is too slow

Application – search algorithms

- Beam search
  - Variable: top 5
  - Expression: top 200
- Anti patterns:
  - object != null

Evaluation 1: Predicting conditions within a project

- Two Java projects: Apache-Math and Joda-Time
- Only the **literal equal** result is considered as **Correct**

Origin: `a > 10` `a >= 11, a > (int) 10.0`

- Math: 1/10 = **509 conditions**
- Time: 1/10 = **194 conditions**
- Precision = Correct / (Correct + Incorrect)

Project	Top 1 Precision	Top 10 Precision	Top 50 Precision
Math12	37.8%	62.2%	69.9%
Time11	48.9%	71.2%	75.5%%
Average	43.5%	66.7%	72.7%

Evaluation 2: L2S in APR

- Dataset: Math and Time in Defects4J benchmark
- In total 133 defects
- Utilizing the same two fix patterns with ACS.
- L2S can fix 11 bugs *correctly*.
- Comparing with the state-of-arts, L2S can fix 4 new bugs

Table 4: Overall Comparison with Existing Techniques (Correct / Incorrect)

Technique	Commons-Math	Joda-Time	Total
L2S-E	9 / 12	2 / 4	11 / 16
ACS	12 / 4	1 / 0	13 / 4
Nopol	1 / 20	0 / 1	1 / 21

Discussion

- L2S can predict patches with higher quality

Math25

The patch of ACS:

```
if(c2 == 0)  // c2 is double
```

The patch of L2S:

```
if(Precision.equals(c2,0))
```

had better to consider precision when comparing with double

- L2S can generate more complex patches comparing with ACS and Nopol

Math33

The patch of L2S:

```
if ( Precision.compareTo(entry, 0d, epsilon) > 0 ) {
```

both ACS and Nopol are unable to generate this patch

Summary and Future Work

- Learning to Synthesize: a framework for estimating a program under a context
- Work-in-progress
- Many future directions
  - Applications
    - Program repair, code generation, test generation, fault localization
  - Can we automate the choice of rule set?
  - What is the effect of different policies for choosing nodes?
  - Can we use better search algorithms?

Thank you！