CSC 2547H: AUTOMATED REASONING WITH MACHINE LEARNING

Xujie Si
Assistant Professor
Department of Computer Science
University of Toronto
Paper presentation

- Grading rubrics
  - Preparation (15%)
    - Sign up on Ed 5%
    - Get feedback from TA 5%
    - Practice Recording 5%
  - Presentation (70% + 15% bonus)
    - Provide the necessary background 10%
    - Explain the problem and main challenges 10%
    - Illustrate the main ideas clearly 15%
    - Show the main results 15% + demo (15% bonus)
    - Limitations / related / future work discussion 10%
    - Finish under time 10% (around 20 minutes depending on the sign-ups)
  - Question Answering (15%)
    - In-class QA (10%)
    - Ed QA (5%)

<table>
<thead>
<tr>
<th>Week</th>
<th>#Sign-ups</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 4: ml4sat</td>
<td>4</td>
</tr>
<tr>
<td>Week 5: ml4smt</td>
<td>3</td>
</tr>
<tr>
<td>Week 6: fm4ml</td>
<td>6</td>
</tr>
<tr>
<td>Week 7: ml4code</td>
<td>2</td>
</tr>
<tr>
<td>Week 8: dl4code</td>
<td>4</td>
</tr>
<tr>
<td>Week 9: dl+logic</td>
<td>4</td>
</tr>
<tr>
<td>Week 10: nv-sym</td>
<td>1</td>
</tr>
</tbody>
</table>
Lecture Overview

• Program Analysis
  - Dynamic Analysis
  - Static Analysis

• Program Synthesis
  - Programming by Examples / Demonstrations
  - Syntax-guided Program Synthesis

<table>
<thead>
<tr>
<th>Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1960</td>
<td>2020</td>
</tr>
</tbody>
</table>

with learning?
Program Analysis

• Given a program, analyze whether it is “good”

• What is “good” or “bad”?
  - Correctness
  - Performance
  - Energy efficiency
  - Memory footprint
  - Fault tolerance
  - Easy to read / maintain
  - Obscure enough to protect IP
  - Side channels (e.g., timing, cache miss, etc.)
  - Liveness, fairness, no crashes (e.g., deadlock, segment fault, etc.)
Dynamic Analysis

- **Software Testing**
  - Unit testing / Integration testing / System testing / Acceptance testing (alpha, beta)
  - Regression testing / compatibility testing
  - White-box testing / black-box testing / gray-box testing
  - Differential testing
  - Fuzzing
  - Mutation testing
  - Delta debugging
  - Code coverage

**DBMS Testing**: 400+ bugs in widely-used DBMS (SQLite, MySQL, MariaDB, PostgreSQL, CockroachDB, and TiDB)

**Project Yin-Yang for SMT Solver Testing**: [Z3/CVC4 bugs: 1,560 (total) / 1,061 (fixed)]
  [Reports: YinYang, OpFuzz, TypeFuzz]

**EMI & SPE Compiler Testing**: [GCC/LLVM bugs: 1,634 (total) / 1,076 (fixed)]
  [Reports: GCC (link1, link2, link3, link4, link5), LLVM (link1, link2, link3, link4, link5)]

https://lcamtuf.coredump.cx/afl/
https://google.github.io/oss-fuzz/
https://people.inf.ethz.ch/suz/
Symbolic Execution

- Assuming there exists at least one input following a given execution path
- Keep track of symbolic constraints along the given path step by step
- Solve constraints
  - SAT $\Rightarrow$ a concrete test case
  - UNSAT $\Rightarrow$ a proof that the given execution path is infeasible

```
def foo(x, y):
    if x < 10:
        y += x
        if y < 15:
            assert(False)  # crash 1
        else:
            assert(x < 2*y)  # crash 2
    return 100
elif x == 1234:
    return 5678
else:
    return x * y
```

$x < 10 \land y_2 = x + y \land y_2 < 15$

$x < 10 \land y_2 = x + y \land y_2 \geq 15 \land x \geq 2 \times y_2$
Dynamic Symbolic Execution

• When constraints become just too complicated
  - Option-1: give up completely (not good 😞)
  - Option-2: Instantiate some symbolic values with concrete values, and move on

```python
1 def foo(x, y, z):
2     if 2**x <= y * sin(y):
3         w = x + z
4         if w < y:
5             assert (w > 0)  # 2^x ≤ y * sin(y) \land w = x + z \land w < y \land w ≤ 0
6     else:
7         assert (y <= z)  # \frac{1}{2} ≤ \frac{\pi}{2} \cdot sin(\frac{\pi}{2}) \land w = (-1) + z \land w < \frac{\pi}{2} \land w ≤ 0
```
Microsoft Zune

Zune bug explained in detail

Devin Coldewey  @techcrunch  /  9:58 PM EST  December 31, 2008

Earlier today, the sound of thousands of Zune owners crying out in terror made ripples across the blogosphere. The response from Microsoft is to wait until tomorrow and all will be well. You’re probably wondering, what kind of bug fixes itself?

Well, I’ve got the code here and it’s very simple, really; if you’ve taken an introductory programming class, you’ll see the error right away.
year = ORIGINYEAR;
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}

static int IsLeapYear(int Year)
{
    int Leap;
    Leap = 0;
    if ((Year % 4) == 0) {
        Leap = 1;
        if ((Year % 100) == 0) {
            Leap = (Year % 400) ? 0 : 1;
        }
    }
    return (Leap);
}
Static Analysis

• Analyze programs without executing them
  - Prevent bugs in the earliest stage (i.e., before any execution happens)
• Compiler warnings and errors
• Linters, static checkers
• Type Checking and Inference
• Program Verification

https://clang-analyzer.llvm.org/available_checks.html

```c
int foo(int length) {
    int x = 0;
    for (int i = 0; i < length; i++)
        x += 1;
    return length/x;
}
```
Type Checking and Inference

• Type safety
  - Minimum requirement of non-buggy code
  - e.g. “hello” + 1.23 does not make sense

• Dynamic typing vs static typing
  - Python, JavaScript, PHP, Ruby, Racket, etc.
  - C/C++, C#, Java, Rust, Go, Standard ML, Ocaml, Haskell, etc.

• Strongly statically typed languages tend to be safer, faster, less annotations, modular ...

• Linear type
  - Rust, Haskell, Idris, Linear ML, etc.

• Dependent type
  - Agda, Coq, Lean, Idris, Dependent ML, etc.

https://openai.com/blog/formal-math/
https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/
Program Verification

• Symbolic Executions
  - Explicitly enumerate and verify each path
  - How about loops / recursions?

• Theorem Proving
  - Hoare logic, separation logic, intuitionistic/constructive logic, etc.

• Abstract Interpretation

• Software Model Checking
Theorem Proving

- Hoare Logic

\[
\{p\} \text{skip} \{p\}
\]

\[
\{p[x \leftarrow t]\} \ x := t \{p\}
\]

\[
\frac{\{p\} \ S_1 \ {r} \quad \{r\} \ S_2 \ {q}}{\{p\} \ S_1; \ S_2 \ {q}}
\]

\[
\frac{\{p \land b\} \ S \ {p}}{\{p\} \ \text{while} \ b \ \text{do} \ S \ {p \land \neg b}}
\]

\[
\frac{\{p_1\} \ S \ {q_1}}{\{p\} \ S \ {q}}
\]

provided \( p \Rightarrow p_1 \) and \( q_1 \Rightarrow q \).


Loop Invariant (Fundamental Challenge)
Theorem Proving

- **Separation Logic**
  - Reason about programs that manipulate pointer data structures
  - Split heaps into disjoint parts
  - Enable scalable compositional reasoning

\[ P \times Q \]

"and, separately"

\[ x \mapsto 1 \times y \mapsto 1 \]

Peter O’hearn, *Separation logic*, CACM 2019
Theorem Proving

• Intuitionistic/Constructive Logic/Dependent Types
  - A theorem is a type
  - Building a proof is essentially constructing an object of that type

Curry–Howard correspondence

https://deepspec.org/main
Abstract Interpretation

• A theory of sound approximation of the semantics of programs
  - Abstract domain $\mathbb{A}$, and Concrete domain $\mathbb{C}$
  - Abstraction function, $\alpha: \mathbb{C} \rightarrow \mathbb{A}$
    \[ x \in \gamma(\alpha(x)) \]
  - Concretization function, $\gamma: \mathbb{A} \rightarrow 2^\mathbb{C}$
    \[ x = \alpha(\gamma(x)) \]

```
def foo(x):
    if x <= 0:
        return 1 - x
    else:
        return x
```

Cousot & Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, POPL 1977
Successful Applications of A.I.

Astrée
https://www.astree.ens.fr/
Program Verification

• Model checking
  - Check whether a finite-state model (FSM) meets a given specification (usually in temporal logic)
  - Earlier successes are in hardware design

• Symbolic model checking
  - Using Binary decision diagram (BDD) to represent all possible states

A hardware bug incurred $475 million loss
• Bounded model checking
  
  - Unroll FSM up to a fixed number of steps
  - Check whether a property is violated using SAT solver
Program Verification

- Software Model Checking
  - Extend model checking to software, which has infinite number of states
Program Verification

- Constrained Horn Clauses

\[ \forall \forall \cdot (\varphi \land p_1(X_1) \land \cdots \land p_k(X_k) \rightarrow h(X)), \text{ for } k \geq 1 \]

```c
main()
{
    int x, y;
    x=1; y=0;
    while(*){
        x=x+y;
        y++;
    }
    assert (x>=y);
}
```

1. \( x = 1 \land y = 0 \rightarrow p(x, y) \) 
2. \( p(x, y) \land x' = x + y \land y' = y + 1 \rightarrow p(x', y') \)

Lecture Overview

• Program Analysis
  - Dynamic Analysis
  - Static Analysis

• Program Synthesis
  - Programming by Examples / Demonstrations
  - Syntax-guided Program Synthesis
Like neural networks, an old idea becomes fashionable again ...
Programming by Examples

Wang et al., *Synthesizing Highly Expressive SQL Queries from Input-Output Examples*, PLDI 2017

https://stackoverflow.com/questions/40015743/jpa-distinct-and-limiting-result-number
Programming by Examples

Let’s say we want to find cheap silver cameras on Amazon ...

![Diagram of web automation process](image)

**Server** → **JavaScript** → **DOM** → **User**

- **Request**
- **Response**
- **JS handler**
- **Hover**
- **Click**
- **Price updated**
- **Scrape price**

---

Excel 2013’s coolest new feature that should have been available years ago”

FlashFill demo

Automating String Processing in Spreadsheets Using Input-Output Examples

Sumit Gulwani
Microsoft Research, Redmond, WA, USA
sumit@microsoft.com

Abstract
We describe the design of a string processing/expression handler for spreadsheets that supports restricted forms of regular expressions, conditions, and loops. The language is expressive enough for most spreadsheet manipulation tasks, including import and export of text files, input validation, and simple data transformations.

POPL Most Influential Paper Award 2021
Any concerns of PBE?

Program synthesis in a nutshell

Given \( \{(i_1, o_1), \ldots, (i_n, o_n)\} \), synthesize a program

Easy!

\[
\begin{align*}
    &\text{if } I = i_1, \text{ then return } o_1 \\
    &\text{if } I = i_2, \text{ then return } o_2 \\
    &\ldots \\
    &\text{if } I = i_n, \text{ then return } o_n
\end{align*}
\]
Syntax-guided Program Synthesis

Logical formula $\varphi(x,y)$

Semantic Specification

Syntactic Specification

Set E of expressions

Search for $e$ in $E$ s.t. $\varphi(x,e(x))$

Synthesizer

Implementation

SyGus
Syntax-Guided Synthesis
SyGuS Problem

- Fix a background theory T: fixes types and operations

- Function to be synthesized: name f along with its type
  - General case: multiple functions to be synthesized

- Inputs to SyGuS problem:
  - Specification $\varphi(x, f(x))$
    - Typed formula using symbols in T + symbol f
  - Set E of expressions given by a context-free grammar
    - Set of candidate expressions that use symbols in T

- Computational problem:
  - Output $e$ in E such that $\varphi[f/e]$ is valid (in theory T)
Theory QF-LIA (Quantifier-free linear integer arithmetic)
Types: Integers and Booleans
Logical connectives, Conditionals, and Linear arithmetic
Quantifier-free formulas

Function to be synthesized: \( f \) (int \( x_1, x_2 \)) : int

Specification: \((x_1 \leq f(x_1, x_2)) \& (x_2 \leq f(x_1, x_2))\)

Candidate Implementations: Linear expressions
\[ \text{LinExp} := x_1 \mid x_2 \mid \text{Const} \mid \text{LinExp} + \text{LinExp} \mid \text{LinExp} - \text{LinExp} \]

No solution exists
SyGuS Example 2

- Theory QF-LIA

- Function to be synthesized: \( f(\text{int } x_1, x_2): \text{int} \)

- Specification: \( (x_1 \leq f(x_1, x_2)) \& (x_2 \leq f(x_1, x_2)) \)

- Candidate Implementations: Conditional expressions without +

  \[
  \text{Term} := x_1 | x_2 | \text{Const} | \text{If-Then-Else (Cond, Term, Term)} \\
  \text{Cond} := \text{Term} \leq \text{Term} | \text{Cond} \& \text{Cond} | \sim \text{Cond} | (\text{Cond})
  \]

- Possible solution:
  \( \text{If-Then-Else } (x_1 \leq x_2, x_2, x_1) \)
How to solve SyGuS problems?

• Enumerative Search
• Constraint Solving (SMT)
• Stochastic Search
Enumerative Search

• Enumerate programs from small to large
• Pruning is important
• Run test cases
• No need to enumerate equivalent sub-expressions
  - But how to avoid that?
• Use pre-defined rules
  - E.g. $A + B = B + A$
• Indistinguishability based on tests

Udupa et al., TRANSIT: specifying protocols with concolic snippets, PLDI 2013

Alur et al., Synthesis Through Unification, CAV 2015
Is it effective?

Benchmarks Solved per Solver per Track

Expression Sizes

<table>
<thead>
<tr>
<th>Solver</th>
<th>Sum ExprSize</th>
<th>Max ExprSize</th>
<th>Avrg ExprSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVC4₂₀₁₇</td>
<td>6193196</td>
<td>1843271</td>
<td>16559.34759</td>
</tr>
<tr>
<td>EUSolver₂₀₁₇</td>
<td>16333</td>
<td>2551</td>
<td>40.62935323</td>
</tr>
<tr>
<td>Euphony</td>
<td>16009</td>
<td>2551</td>
<td>44.34626039</td>
</tr>
</tbody>
</table>

Stochastic Search

- Start with a random program
- Mutate randomly (MCMC sampling)
- Stop when finding the correct program

Search($J$: Initial candidate)
Returns: A candidate $C$ with $c_V(C) = 0$.

1. $C := J$
2. while $c_V(C) \neq 0$ do
3. $m := \text{SampleMove}(\text{rand}())$
4. $C' := m(C)$
5. $c_o := c_V(C)$, $c_n := c_V(C')$
6. if $c_n < c_o$ or $e^{-\gamma(c_n-c_o)} > \frac{\text{rand}()}{\text{RANDMAX}}$ then
7. $C := C'$
8. end if
9. end while
10. return $C$
1 # gcc -O3
2
3 .L0:
4 movq rsi, r9
5 movl ecx, ecx
6 shrq 32, rsi
7 andl 0xfffffffff, r9d
8 movq rcx, rax
9 movl edx, edx
10 imulq r9, rax
11 imulq rdx, r9
12 imulq rsi, rdx
13 imulq rsi, rcx
14 addq rdx, rax
15 jae .L2
16 movabsq 0x100000000, rdx
17 addq rdx, rcx
18 .L2:
19 movq rax, rsi
20 movq rax, rdx
21 shrq 32, rsi
22 salq 32, rdx
23 addq rsi, rcx
24 addq r9, rdx
25 adcq 0, rcx
26 addq r8, rdx
27 adcq 0, rcx
28 addq rdi, rdx
29 adcq 0, rcx
30 movq rcx, r8
31 movq rdx, rdi

1 # STOKE
2
3 .L0:
4 shlq 32, rcx
5 movl edx, edx
6 xorq rdx, rcx
7 movq rcx, rax
8 mulq rsi
9 addq r8, rdi
10 adcq 0, rdx
11 addq rdi, rax
12 adcq 0, rdx
13 movq rdx, r8
14 movq rax, rdi

Schkufza et al, Stochastic Superoptimization, ASPLOS 2013
Success Example

Figure 5: Speedups by benchmark. For each benchmark, the speedup over the original NaCl library is shown. The bars correspond to the optimization experiment, the translation experiment, and the best rewrite we verified. The ‘optimization mode’ much more reliably produces a verifiable result, but ‘translation mode’ sometimes offers significant improvements.

5. Evaluation

We use 13 libc string functions from the newlib library shipped with Google Native Client to evaluate our extensions to STOKE. We performed all experiments on machines with two Intel Xenon E5-2667v2 3.3GHz processors and 256GB of RAM.

We evaluate our work in three categories. First, we demonstrate that we can optimize these benchmarks and achieve formally verified NaCl code with a median and average speedup of 25%. Then, we compare the baseline implementation with our new system that uses the bounded verifier. Finally, we compare the performance of the alias relationship mining to the flat memory model.

5.1 Experiment Setup

Our goal is to improve the performance of each of the 13 libc string functions and prove correctness of the optimized code. For each benchmark we perform two experiments, optimization and translation. In optimization mode, we initialize the rewrite with the code shipped with NaCl and run STOKE to improve its performance while maintaining compliance with the NaCl rules. In translation mode, the rewrite is initialized with code that does not comply with NaCl rules and STOKE transforms it into well-formed NaCl code. For each benchmark, we assembled test cases from randomly generated strings.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Target LOC</th>
<th>Best LOC</th>
<th>Best Speedup</th>
<th>Search Time (min)</th>
<th>DDEC Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wcpcpy</td>
<td>40</td>
<td>13</td>
<td>48%</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>wcslen</td>
<td>43</td>
<td>47</td>
<td>97%</td>
<td>78</td>
<td>89</td>
</tr>
<tr>
<td>wmemset</td>
<td>47</td>
<td>47</td>
<td>0%</td>
<td>29</td>
<td>45</td>
</tr>
<tr>
<td>wcsnlen</td>
<td>94</td>
<td>51</td>
<td>2%</td>
<td>61</td>
<td>83</td>
</tr>
<tr>
<td>wmemcmp</td>
<td>91</td>
<td>77</td>
<td>47%</td>
<td>360</td>
<td>302</td>
</tr>
<tr>
<td>wcschr</td>
<td>87</td>
<td>28</td>
<td>2%</td>
<td>61</td>
<td>5</td>
</tr>
<tr>
<td>strxfrm</td>
<td>99</td>
<td>38</td>
<td>0%</td>
<td>81</td>
<td>414</td>
</tr>
<tr>
<td>wcscmp</td>
<td>108</td>
<td>29</td>
<td>47%</td>
<td>38</td>
<td>586</td>
</tr>
<tr>
<td>wmemchr</td>
<td>132</td>
<td>75</td>
<td>2%</td>
<td>67</td>
<td>30</td>
</tr>
<tr>
<td>wcscpy</td>
<td>35</td>
<td>40</td>
<td>25%</td>
<td>276</td>
<td>252</td>
</tr>
<tr>
<td>wcscat</td>
<td>89</td>
<td>90</td>
<td>26%</td>
<td>360</td>
<td>46</td>
</tr>
<tr>
<td>strcpy</td>
<td>70</td>
<td>63</td>
<td>30%</td>
<td>360</td>
<td>415</td>
</tr>
<tr>
<td>wcsrcchr</td>
<td>178</td>
<td>178</td>
<td>0%</td>
<td>30</td>
<td>15</td>
</tr>
</tbody>
</table>

Churchill et al, Sound Loop Superoptimization for Google Native Client, ASPLOS 2017