Type Assisted Synthesis of Programs with Algebraic Data Types
Jeewana Priya Ihal
Advisor: Armando Solar-Lezama
MIT CSAIL

OBJECTIVES
• Synthesize complicated programs involving pattern matching and algebraic data types (ADT) from simple templates
• Particularly, programs that transform data structures like desugaring languages and optimizing abstract syntax trees (AST) as used in compilers
• Make synthesis enabled tools both more efficient and easier to develop

Example 1 - Desugaring a simple language
Goal: Synthesize a desugar function from srcAST to dstAST

ADT Definitions

```latex
\begin{aligned}
\text{dstAST} &::= \text{BinaryS} \quad \text{dstAST} \quad \text{FalseS} \quad \text{TrueS} \\
\text{srcAST} &::= \text{NumS} \\
\end{aligned}
```

Specification

```latex
\begin{aligned}
\text{interpt}(s) &::= \text{interpt}(\text{desugar}(s)) \\
\end{aligned}
```

Template of desugar function

```latex
\begin{aligned}
\text{dstAST} &::= \text{desugar}(s) \\
\text{dstAST} &::= \text{srcAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\end{aligned}
```

Output

```latex
\begin{aligned}
\text{dstAST} &::= \text{desugar}(s) \\
\text{dstAST} &::= \text{srcAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\text{dstAST} &::= \text{dstAST} \circ \text{dstAST} \\
\end{aligned}
```

Example 2 - AST optimizations
Goal: Given an abstract syntax tree node, synthesize an optimized node and the corresponding predicate

```latex
\begin{aligned}
\text{Example optimization rule} &::= \text{b} \Rightarrow \text{d} \\
\end{aligned}
```

Type Directed Reduction

- Need to perform type inference and reduction in tandem
- This requires type information to be propagated both top-down and bottom-up
- Use bi-directional rules that make this flow of information explicit

Symbolic Execution

- Inline function calls, unrolls loops and creates a formula to encode to the SAT solver
- Uses Counter Example Guided Inductive Synthesis (CEGIS)

Challenges: scalability and encoding algebraic data types to SAT solver

Optimizations to improve scalability

1. Merging recursive calls with mutually exclusive path conditions to avoid inlining blowup
2. Abstacting recursive calls of function to be synthesized by assuming the specification is true

Encoding algebraic data types to SAT

- Use recursive tuples that can leverage the immutability of algebraic data types
- Uniary based encoding for recursive tuples in SAT solver

RESULTS

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Runtime (sec)</th>
<th>Distinct choices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion into a binary tree using examples</td>
<td>7.44</td>
<td>2^72</td>
</tr>
<tr>
<td>Insertion into a binary tree using behavioral constraints</td>
<td>18.42</td>
<td>2^72</td>
</tr>
<tr>
<td>Simple language desugaring</td>
<td>36.35</td>
<td>2^110</td>
</tr>
<tr>
<td>Simple language desugaring with state</td>
<td>677.19</td>
<td>2^141</td>
</tr>
<tr>
<td>Boolean to Lambda Calculus</td>
<td>114.14</td>
<td>2^541</td>
</tr>
<tr>
<td>Pairs to Lambda Calculus</td>
<td>683.55</td>
<td>2^183</td>
</tr>
<tr>
<td>AST optimizations</td>
<td>163.09</td>
<td>2^182</td>
</tr>
<tr>
<td>Type constraints for Lambda Calculus with examples</td>
<td>10.23</td>
<td>2^149</td>
</tr>
<tr>
<td>Type constraints for Lambda Calculus using behavioral constraints</td>
<td>496.12</td>
<td>2^149</td>
</tr>
</tbody>
</table>

REFERENCES