Type Assisted Synthesis of Programs with Algebraic Data Types

Jeevana Priya Inala
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 language constructs 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

```
adt srcAST {
  NumS { int v; }
  TrueS { }
  FalseS { }
  BinaryS { opcode op; srcAST a; srcAST b; }
  BetweenS{ srcAST a; srcAST b; srcAST c; }
}
```

```
adt dstAST {
  NumD { int v; }
  BoolD {bit v;}
  BinaryD { opcode op; srcAST a; srcAST b; }
}

adt opcode{ PlusOp{} MinusOp{} AndOp{} OrOp{}
LtOp{} GtOp{} }
```

Specification

```
interpretSrc(s)  == interpretDst(desugar(s))
```

Template of desugar function

```
dstAST desugar(srcAST s) {
 if (s == null) return null;
 switch(s){
 repeat_case: {
   dstAST a = desugar(s.??);
   dstAST b = desugar(s.??);
   dstAST c = desugar(s.??);
   return ??(3, {a, b, c, s.??});

}}} 
```

LOC: 7

Output

```
dstAST desugar(srcAST s) {
 if (s == null) return null;
 switch(s){
 case NumS: { return new NumD(v = s.v); }
 .....
 case BetweenS: {
    dstAST a = desugar(s.a);
    dstAST b = desugar(s.b);
    dstAST c = desugar(s.c);
    return new BinaryD(op = new AndOp(),
          a = new BinaryD(op = new LtOp(), a = a, b = b),
          b = new BinaryD(op = new LtOp(), a = b, b = c)); }
 .....
}}
```

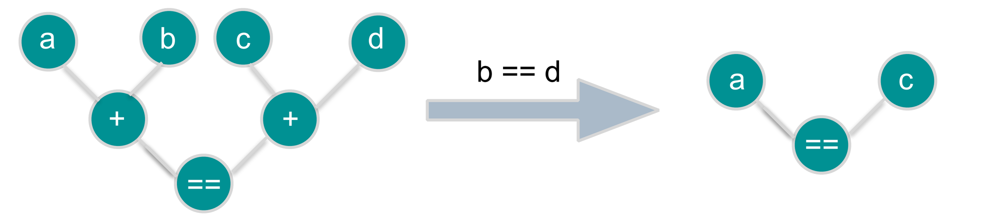
LOC: 22

Synthesis Time: 36s
No. of possible functions from template ~ 2^{110}

Example 2 - AST optimizations

Goal: Given an abstract syntax tree node, synthesize an optimized node and the corresponding predicate

Example optimization rule

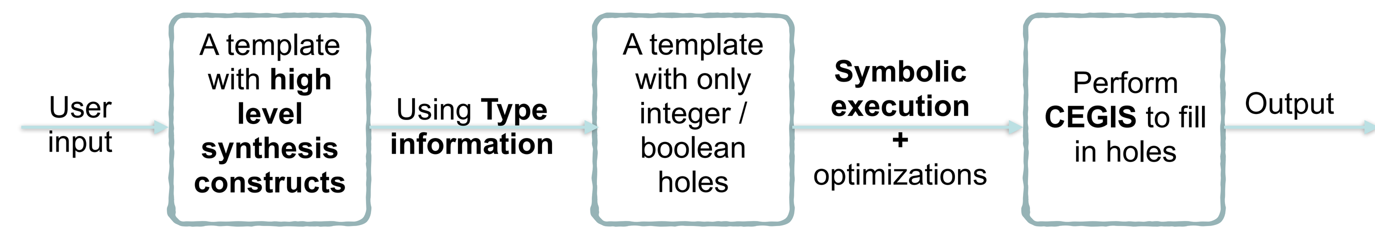


Template program and specification

```
AST original = new Equal(a = new Plus(a= new Var(id = 0), b = new Var(id = 1)),
                          b = new Plus(a = new Var(id = 2), b= new Var(id =3)));

harness AST optimize (int[4] assignment) {
    AST predicate = ??(3, {});  // Find a predicate

    // Make sure that there exists atleast one assignment that satisfies the predicate
    int[4] tmp = ??;
    assert( isSatisfied(predicate, tmp) );

    if (isSatisfied(predicate, assignment)) { // If predicate is satisfied
        AST optimized = ??(3, {});

        // Assert that both versions yield the same output when run on the given assignment
        assert(run(original, assignment) === run(optimized, assignment)));
        // Assert that the optimized node is actually smaller in size
        assert(count(optimized) < count(original));
        return optimized;
    }
    return original;
}
```

OUR APPROACH

- Template based synthesis like Sketch [1]
- Sketch is a C-like language that supports integer and boolean holes
- Performs symbolic execution using a SAT solver to fill in these holes



NEW SYNTHESIS CONSTRUCTS

Designed on top of integer and boolean holes so that it is

- Easy for users to write the template
- Easy for the solver to synthesize correct solution

Construct	Syntax
Switch and repeat_case	switch(e) { repeat_case: body }
Field selector hole	e.??
Generalized Unknown Constructor	??(k, { $e_1 \cdots e_m$ })
Unknown Constructor	new ??($l_t = e_t$)

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 [2]

General form of reduction rule

$$\frac{T = \{\tau_0 \ldots \tau_i\}}{\Gamma \vdash e \xrightarrow{T} \{e_0 \ldots e_k\}}$$

Reduction rule for Variable

$$\frac{x : \tau \in \Gamma \quad \tau \in T}{\Gamma \vdash x \xrightarrow{T} \{x\}} \qquad \frac{x : \tau \in \Gamma \quad \tau \notin T}{\Gamma \vdash x \xrightarrow{T} \emptyset}$$

Reduction rule for Field Selector Hole

$$\frac{\Gamma \vdash e \xrightarrow{T'} \{e_0 \ldots e_k\} \;\; where\ T^{'} = \{\tau \mid \tau\ has\ a\ field\ l : \tau_i\ and\ \tau_i \in T\ \}}{\Gamma \vdash e.?? \xrightarrow{T} \{e_i.l_j \mid e_i.l_j : \tau \;\; j \in [0,k]\ and\ \tau \in T\}}$$

Reduction rule for switch and repeat_case

$$\frac{\Gamma = (\Gamma^{'}; x : \sum name_i \{l^i_1 : \tau^i_1 \ldots l^i_n : \tau^i_n\}) \quad (\Gamma^{'}; x : \{l^i_1 : \tau^i_1 \ldots l^i_n : \tau^i_n\}) \vdash e \xrightarrow{\{\tau\}} E_i = \{e^0_i \ldots e^{k_i}_i\}}{\Gamma \vdash switch\,(x)\,\{\ repeat\_case : e\ \} \xrightarrow{\{\tau\}} \{switch\,(x)\,\{\ case\ name_i : choose(E_i)\}\}}$$

SYMBOLIC EXECUTION

- Inlines 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

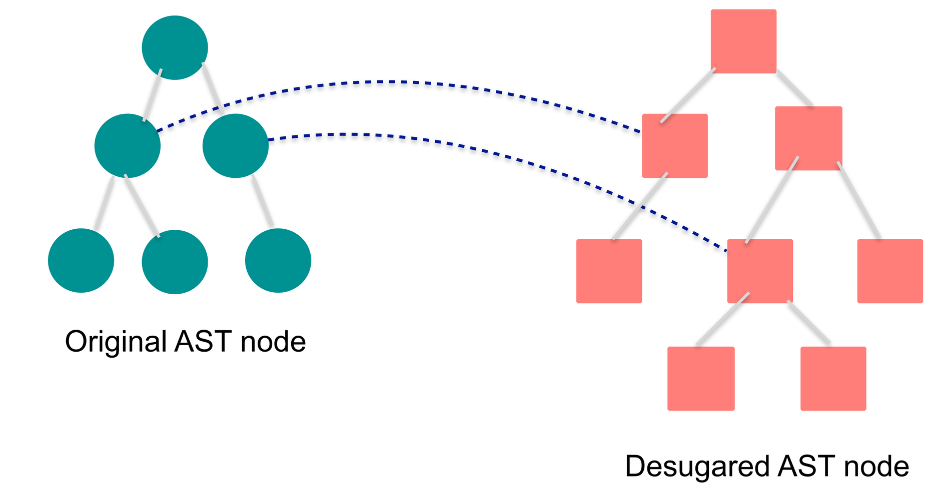
1. Merging recursive calls with mutually exclusive path conditions to avoid inlining blowup

```
if (w)
    x = f(a, b);
else
    y = f(c, d);
```

→

```
t = f(w?a:c, w?b:d);
if (w)
    x = t;
else
    y = t;
```

2. Abstracting recursive calls of function to be synthesized by assuming the specification is true



- Synthesizer needs to reason about one variant at a time
- Runtime is linear in number of variants instead of exponential

Encoding algebraic data types to SAT

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

RESULTS

Benchmark	Runtime (sec)	Distinct choices
Insertion into a binary tree using examples	7.44	2^72
Insertion into a binary tree using behavioral constraints	18.42	2^72
Simple language desugaring	36.36	2^110
Simple language desugaring with state	577.19	2^141
Booleans to Lambda Calculus	114.14	2^541
Pairs to Lambda Calculus	683.55	2^183
AST optimatizations	163.09	2^162
Type constraints for Lambda Calculus with examples	10.23	2^149
Type constraints for Lambda Calculus using behavioral constraints	496.12	2^149

REFERENCES

1. A. Solar-Lezama. Program Synthesis By Sketching. PhD thesis, EECS Dept., UC Berkeley, 2008.
2. B. C. Pierce and D. N. Turner. Local type inference. ACM Trans. Program. Lang. Syst., 22(1): 1-44, Jan. 2000.