PROGRAM SYNTHESIS

RUZICA PISKAC
YALE UNIVERSITY
Search through all possible ways to fill the hole “??” such that the specification is satisfied:

Find $c$ such that:

$$\forall x. \ (x \ast c = x + x)$$

Equivalently:

$$\exists c. \forall x. \ (x \ast c = x + x)$$

In general, every sketching problem can be converted to solving a formula of the form:

$$\exists c_1 c_2 \ldots \forall x_1 x_2. \ \varphi(c_1, c_2, \ldots, x_1, x_2 \ldots)$$
RECIPE FOR CONSTRAINT BASED SYNTHESIS

1. Convert the synthesis problem $P$ to a formula $\varphi$
2. Solve $\varphi$ using a constraint solver
3. Map solution from constraint solver back to the synthesis problem $P$

Types of constraint solvers:
- SAT Solvers (Boolean Satisfiability)
- SMT Solvers (Satisfiability Modulo Theories)
- Solvers for Quantified Formulas (QBF or CEGIS solvers)
The classical formulation of the program-synthesis problem is to find a program that meets a correctness specification given as a logical formula. Recent work on program synthesis and program optimization illustrates many potential benefits of allowing the user to supplement the logical specification with a syntactic template that constrains the space of allowed implementation. The motivation is twofold. First, narrowing the space of implementations makes the synthesis problem more tractable. Second, providing a specific syntax can potentially lead to better optimizations.

The Problem

A Syntax-Guided Synthesis problem (SyGuS, in short) is specified with respect to a background theory $T$, such as Linear-Integer-Arithmetic (LIA), that fixes the types of variables, operations on types, and their interpretation.

To synthesize a function $f$ of a given type, the input consists of two constraints: (1) a semantic constraint given as a formula $\varphi$ built from symbols in theory $T$ along with $f$, and (2) a syntactic constraint given as a (possibly infinite) set $E$ of expressions from $T$ specified by a context-free grammar.

The computational problem then is to find an implementation for the function $f$, i.e. an expression $e \in E$ such that the formula $\varphi[f \leftarrow e]$ is valid.

The Competition

The SyGuS competition (SyGuS-Comp) will allow solvers for syntax-guided synthesis problems to compete on a large collection of benchmarks. The motivation behind the competition is to propagate and advance research and tools on the subject.

SyGuS: SYNTAX GUIDED SYNTHESIS

- 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 of expressions given by a context-free grammar $G$
  - Set of candidate expressions that use symbols in $T$
- Computational problem
  - Output $e$ from grammar $G$, such that $\varphi [e \rightarrow f]$ is valid (in theory $T$)

Syntax-guided synthesis; FMCAD’13
Alur, Bodik, Juniwal, Martin, Raghothaman, Seshia, Singh, Solar-Lezama, Torlak, Udupa
SYNTAX-GUIDED PROGRAM SYNTHESIS

SLIDES BY RAJEEV ALUR AND THE EXCAPE: EXPEDITION TEAM
1. PROGRAMMING BY EXAMPLES (PBE)

Desired program P: bit-vector transformation that resets rightmost substring of contiguous 1’s to 0’s

1. P should be constructed from standard bit-vector operations
   \[ |, \&, \sim, +, -, <<, >>, 0, 1, \ldots \]

2. P specified using input-output examples
   \[
   \begin{align*}
   00101 & \rightarrow 00100 \\
   01010 & \rightarrow 01000 \\
   10110 & \rightarrow 10000 
   \end{align*}
   \]

Desired solution:
\[
x \& ( 1 + (x \mid (x-1) ) )
\]
FLASHFILL: PBE IN PRACTICE

REF: GULWANI (POPL 2011)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>(425)-706-7709</td>
<td>425-706-7709</td>
</tr>
<tr>
<td>510.220.5586</td>
<td>510-220-5586</td>
</tr>
<tr>
<td>1 425 235 7654</td>
<td>425-235-7654</td>
</tr>
<tr>
<td>425 745-8139</td>
<td>425-745-8139</td>
</tr>
</tbody>
</table>

Wired: Excel is now a lot easier for people who aren’t spreadsheet- and chart-making pros. The application’s new Flash Fill feature recognizes patterns, and will offer auto-complete options for your data. For example, if you have a column of first names and a column of last names, and want to create a new column of initials, you’ll only need to type in the first few boxes before Excel recognizes what you’re doing and lets you press Enter to complete the rest of the column.
Task: Collect the max vals below 50 for all oid groups in T2 and join them with T1.

T1
- id: 1, date: 12/25, uid: 1
- id: 2, date: 11/21, uid: 3
- id: 4, date: 12/24, uid: 2

T2
- oid: 1, val: 30
- oid: 1, val: 10
- oid: 1, val: 10
- oid: 2, val: 50
- oid: 2, val: 10

Out
- oid: 1, date: 12/25, uid: 1, oid val MaxVal: 1, 30
- oid: 4, date: 12/24, uid: 2, oid val MaxVal: 2, 10

Select *
From (Select oid, Max(val)
    From T2
    Where val < 50
    Group By oid) T3
Join T1
On T3.oid = T1.uid

Constants = { 50 } 
AggrFunc = { Max, Min }
2. PROGRAM OPTIMIZATION

Can regular programmers match experts in code performance?
  Improved energy performance in resource constrained settings
  Adoption to new computing platforms such as GPUs

Opportunity: Semantics-preserving code transformation

Possible Solution: Superoptimizing Compiler
  Structure of transformed code may be dissimilar to original
Given a program \( P \), find a “better” equivalent program \( P' \)

\[
\text{average (bitvec}[32]\ x, y) \{
  \text{bitvec}[64] x1 = x;
  \text{bitvec}[64] y1 = y;
  \text{bitvec}[64] z1 = (x1+y1)/2;
  \text{bitvec}[32] z = z1;
  \text{return } z
\}
\]

Find equivalent code without extension to 64 bit vectors

\[
\text{average } (x, y) = (x \text{ and } y) + [(x \text{ xor } y) \text{ shift-right 1 }]
\]
The program requires 3 changes:

- In the return statement `return deriv` in line 5, replace `deriv` by `[0]`.
- In the comparison expression `(poly[e] == 0)` in line 7, change `(poly[e] == 0)` to `False`.
- In the expression `range(0, len(poly))` in line 6, replace `0` by `1`.
4. AUTOMATIC INVARIANT GENERATION

SelectionSort(int A[],n) {
    i := 0;
    while(i < n-1) {
        v := i;
        j := i + 1;
        while (j < n) {
            if (A[j]<A[v])
                v := j;
            j++;
        }
        swap(A[i], A[v]);
        i++;
    }
    return A;
}

post: ∀k : 0 ≤k<n ⇒ A[k]≤A[k + 1]
SelectionSort(int A[], n) {
    i := 0;
    while (i < n - 1) {
        v := i;
        j := i + 1;
        while (j < n) {
            if (A[j] < A[v])
                v := j;
            j++;
        }
        swap(A[i], A[v]);
        i++;
    }
    return A;
}

post: ∀k: 0 ≤ k < n ⇒ A[k] ≤ A[k + 1]
**TEMPLATE-BASED AUTOMATIC INARIANT GENERATION**

SelectionSort(int A[], n) {
    i := 0;
    while (i < n - 1) {
        v := i;
        j := i + 1;
        while (j < n) {
            if (A[j] < A[v])
                v := j;
            j++;
        }
        swap(A[i], A[v]);
        i++;
    }
    return A;
}

**Invariant:**

- $\forall k1, k2. \ 0 \leq k1 < k2 < n \land k1 < i \Rightarrow A[k1] \leq A[k2]$

- $\forall k1, k2. \ 0 \leq k1 < k2 < n \land k1 < i \Rightarrow A[k1] \leq A[k2] \land (\forall k. \ i1 \leq k < j \land k \geq 0 \Rightarrow A[v] \leq A[k])$

**post:** $\forall k. \ 0 \leq k < n \Rightarrow A[k] \leq A[k + 1]$
SYNTAX-GUIDED PROGRAM SYNTHESIS

Rich variety of projects in programming systems and software engineering

1. Programming by examples
2. Program superoptimization
3. Automatic program repair
4. Template-guided invariant generation

Computational problem at the core of all these synthesis projects:

Find a program that meets given syntactic and semantic constraints
CLASSICAL PROGRAM SYNTHESIS

Specification
“What”

Synthesizer

Implementation
“How”

Logical relation $\varphi(x,y)$ among input $x$ and output $y$

Constructive proof of
Exists $f$. For all $x$. $\varphi(x,f(x))$

Function $f(x)$ such that
$\varphi(x,f(x))$
SYNTAX-GUIDED PROGRAM SYNTHESIS

Logical formula $\varphi(x,y)$

Semantic Specification

Syntactic Specification

Synthesizer

Implementation

Set $E$ of expressions

Search for $e$ in $E$ such that $\varphi(x,e(x))$
SYNTAX-GUIDED SYNTHESIS: FORMALIZATION
SYNTAX-GUIDED PROGRAM SYNTHESIS

- Find a program snippet $e$ such that
  1. $e$ is in a set $E$ of programs (syntactic constraint)
  2. $e$ satisfies logical specification $\varphi$ (semantic constraint)

- Core computational problem in many synthesis tools/applications

Can we formalize and standardize this computational problem?

Inspiration: Success of SMT solvers in formal verification
SMT: SATISFIABILITY MODULO THEORIES

- Computational problem: Find a satisfying assignment to a formula
  - Boolean + Int types, logical connectives, arithmetic operators
  - Bit-vectors + bit-manipulation operations in C
  - Boolean + Int types, logical/arithmetic ops + Uninterpreted functs

- “Modulo Theory”: Interpretation for symbols is fixed
  - Can use specialized algorithms (e.g. for arithmetic constraints)

Little Engines of Proof

SAT; Linear arithmetic; Congruence closure
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 $T$)

Syntax-guided synthesis; FMCAD’13

with Bodik, Juniwal, Martin, Raghothaman, Seshia, Singh, Solar-Lezama, Torlak, Udupa
SYGUS EXAMPLE 1

- 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
  \( \text{LinExp} := x_1 | x_2 | \text{Const} | \text{LinExp} + \text{LinExp} | \text{LinExp} - \text{LinExp} \)

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

- Possible solution:  
  \( \text{If-Then-Else (} x_1 \leq x_2, x_2, x_1 \) \)
FROM smt-lib TO synth-lib

(set-logic LIA)
(synth-fun max2 ((x Int) (y Int)) Int
  ((Start Int (x y 0 1
  (+ Start Start)
  (- Start Start)
  (ite StartBool Start Start))))
  (StartBool Bool ((and StartBool StartBool)
  (or StartBool StartBool)
  (not StartBool)
  (≤ Start Start))))
(declare-var x Int)
(declare-var y Int)
(constraint (≤ x (max2 x y)))
(constraint (≤ y (max2 x y)))
(constraint (or (= x (max2 x y)) (= y (max2 x y))))
(check-synth)
Invariant Generation as SYGUS

- Goal: Find inductive loop invariant automatically
- Function to be synthesized
  \[ \text{Inv} (\text{bool } x, \text{bool } z, \text{int } a, \text{int } b) : \text{bool} \]
- Compile loop-body into a logical predicate
  \[ \text{Update}(x,y,z,a,b,c, x',y',z',a',b',c') \]
- Specification:
  \[ (\text{Inv} \land \text{Update} \land \text{Test}') \Rightarrow \text{Inv}' \]
  \[ \text{Pre} \Rightarrow \text{Inv} \land (\text{Inv} \land \neg \text{Test} \Rightarrow \text{Post}) \]
- Template for set of candidate invariants
  \[ \text{Term} := a \mid b \mid \text{Const} \mid \text{Term} + \text{Term} \mid \text{If-Then-Else} (\text{Cond}, \text{Term}, \text{Term}) \]
  \[ \text{Cond} := x \mid z \mid \text{Cond} \land \text{Cond} \mid \neg \text{Cond} \mid (\text{Cond}) \]
SOLVING SYGUS
SOLVING SYGUS

Is SyGuS same as solving SMT formulas with quantifier alternation?

SyGuS can sometimes be reduced to Quantified-SMT, but not always

- Set $E$ is all linear expressions over input vars $x, y$
  - SyGuS reduces to $\exists a, b, c. \forall X. \varphi \left[ f/ ax+by+c \right]$
- Set $E$ is all conditional expressions
  - SyGuS cannot be reduced to deciding a formula in LIA

Syntactic structure of the set $E$ of candidate implementations can be used effectively by a solver

Existing work on solving Quantified-SMT formulas suggests solution strategies for SyGuS
SYGUS AS ACTIVE LEARNING

Initial examples I

Search Algorithm

Candidate Expression

Verification Oracle

Counterexample

 Fail

Success

Concept class: Set E of expressions

Examples: Concrete input values
COUNTEREXAMPLE-GUIDED INDUCTIVE SYNTHESIS
SOLAR-LEZAMA ET AL (ASPLOS'06)

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

- Set E: All expressions built from \(x_1, x_2, 0, 1,\) Comparison, If-Then-Else

\[ I = \{ \} \]

\[ f(x_1, x_2) = x_1 \]

Search Algorithm

Verification Oracle

Candidate

Counterexample

\( (x_1=0, x_2=1) \)
CEGIS EXAMPLE

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

- Set E: All expressions built from \(x_1, x_2, 0, 1,\) Comparison, If-Then-Else

\[ I = \{ (x_1 = 0, x_2 = 1) \} \]

Candidate
\[ f(x_1, x_2) = x_2 \]

Counterexample
\[ (x_1 = 1, x_2 = 0) \]
CEGIS EXAMPLE

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

- Set E: All expressions built from \(x_1, x_2, 0, 1,\) Comparison, If-Then-Else

\[\{(x_1 = 0, x_2 = 1)\]
\[(x_1 = 1, x_2 = 0)\]
\[(x_1 = 0, x_2 = 0)\]
\[(x_1 = 1, x_2 = 1)\}\]

Candidate \(\text{ITE}(x_1 \leq x_2, x_2, x_1)\)

Search Algorithm

Verification Oracle

Success
Goal: Find $f$ in $E$ such that for all $x$ in $D$, $\varphi(x, f)$ holds

$I = \{ \}; /* Interesting set of inputs */$

Repeat

Learn: Find $f$ in $E$ such that for all $x$ in $I$, $\varphi(f, x)$ holds

Verify: Find $x$ in $D$ such that $\varphi(f, x)$ does not hold

If so, add $x$ to $I$

Else, return $f$
SYGUS SOLUTIONS

- CEGIS approach (Solar-Lezama et al, ASPLOS’08)
- Similar strategies for solving quantified formulas and invariant generation

- Initial learning strategies based on:
  1. Enumerative (search with pruning): Udupa et al (PLDI’13)
  2. Symbolic (solving constraints): Gulwani et al (PLDI’11)
1. ENUMERATIVE SEARCH

- Given:
  - Specification $\varphi(x, f(x))$
  - Grammar for set E of candidate implementations
  - Finite set I of inputs

  Find an expression $e(x)$ in E s.t. $\varphi(x, e(x))$ holds for all $x$ in I

- Attempt 0: Enumerate expressions in E in increasing size till you find one that satisfies $\varphi$ for all inputs in I

- Attempt 1: Pruning of search space based on:
  - Expressions $e_1$ and $e_2$ are equivalent if
    - $e_1(x) = e_2(x)$ on all $x$ in I
  - Only one representative among equivalent subexpressions needs to be considered for building larger expressions
ILLUSTRATING PRUNING

- Spec: \((x_1 < f(x_1, x_2)) \& (x_2 < f(x_1, x_2))\)
- Grammar: \(E := x_1 \mid x_2 \mid 0 \mid 1 \mid E + E\)
- \(I = \{(x_1=0, x_2=1)\}\)
- Find an expression \(f\) such that \((f(0,1) > 0) \& (f(0,1) > 1)\)
2. SYMBOLIC SEARCH

- Use a constraint solver for both synthesis and verification steps

- Each production in the grammar is thought of as a component. Input and Output ports of every component are typed.

- A well-typed loop-free program comprising these component corresponds to an expression DAG from the grammar.
SYMBOLIC ENCODING

- Start with a library consisting of some number of occurrences of each component.

- Synthesis Constraints:
  Shape is a DAG, Types are consistent
  Spec \( \varphi[f/e] \) is satisfied on every concrete input in \( I \)

- Use an SMT solver (Z3) to find a satisfying solution.

- If synthesis fails, try increasing the number of occurrences of components in the library in an outer loop.
3. STOCHASTIC SEARCH

- Idea: Find desired expression \( e \) by probabilistic walk on graph where nodes are expressions and edges capture single-edits

- Metropolis-Hastings Algorithm: Given a probability distribution \( P \) over domain \( X \), and an ergodic Markov chain over \( X \), samples from \( X \)

- Fix expression size \( n \). \( X \) is the set of expressions \( E_n \) of size \( n \). \( P(e) \propto \text{Score}(e) \) (“Extent to which \( e \) meets the spec \( \phi \)”)

- For a given set Examples, \( \text{Score}(e) = \exp(-0.5 \text{Wrong}(e)) \), where \( \text{Wrong}(e) = \text{No of inputs in Examples for which } ~ \phi [f/e] \)

- \( \text{Score}(e) \) is large when \( \text{Wrong}(e) \) is small. Expressions \( e \) with \( \text{Wrong}(e) = 0 \) more likely to be chosen in the limit than any other expression
STOCHASTIC SEARCH

- Initial candidate expression $e$ sampled uniformly from $E_n$
- When $\text{Score}(e) = 1$, return $e$
- Pick node $v$ in parse tree of $e$ uniformly at random. Replace subtree rooted at $e$ with subtree of same size, sampled uniformly
- With probability $\min\{1, \frac{\text{Score}(e')}{\text{Score}(e)}\}$, replace $e$ with $e'$
- Outer loop responsible for updating expression size $n$
PART IV

SYGUS COMPETITION AND EVOLUTION
SMT SUCCESS STORY

SMT-LIB Standardized Interchange Format (smt-lib.org)
Problem classification + Benchmark repositories
LIA, LIA_UF, LRA, QF_LIA, ...

+ Annual Competition (smt-competition.org)
SYGUS COMPETITION

Program optimization

Program repair

Programming by examples

Invariant generation

SYNTH-LIB Standardized Interchange Format
Problem classification + Benchmark repository
+ SyGuS-COMP (Competition for solvers) held since FLoC 2014

Techniques for Solvers:
Learning, Constraint solvers, Enumerative/stochastic search

Collaborators: D. Fisman, S. Padhi, A. Reynolds, R. Singh, A. Solar-Lezama, A. Udupa
SYGUS PROGRESS

- Over 2000 benchmarks
  - Hacker’s delight
  - Invariant generation (based on verification competition SV-Comp)
  - FlashFill (programming by examples system from Microsoft)
  - Synthesis of attack-resilient crypto circuits
  - Program repair
  - Motion planning
  - ICFP programming competition

- Special tracks for competition
  - Invariant generation
  - Programming by examples
  - Conditional linear arithmetic

- New solution strategies and applications
SyGuS Conclusions

- **Problem definition**
  - Syntactic constraint on space of allowed programs
  - Semantic constraint given by logical formula

- **Solution strategies**
  - Counterexample-guided inductive synthesis
  - Search in program space + Verification of candidate solutions

- **Applications**
  - Programming by examples
  - Program repair/optimization with respect to syntactic constraints

- **Annual competition (SyGuS-comp)**
  - Standardized interchange format + benchmarks repository
USING SYNTHESIS FOR MODULAR VERIFICATION

© JOINT WORK WITH WILLIAM HALLAHAN AND RANJIT JHALA
map :: (a -> b) -> xs: [a] -> { ys:[b] | size xs == size ys }
map f [] = []
map f (x:xs) = f x: map f xs

8 | map f (x:xs) = map f xs
   ^^^^^^^^^
Inferred type
VV : {v : [a] | size xs == size v 
&& size v >= 0}
not a subtype of Required type
VV : {VV : [a] | size ?a == size VV}
In Context
xs : {v : [a] | size v >= 0}
MODULAR VERIFICATION

add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)

incr :: x:Int -> { y:Int | y > x }
incr x = x + 1

To verify a caller, modular verifiers use callee’s specification

Error: Liquid Type Mismatch
5 | add2 x = incr (incr x)
   ^^^^^^^^^^^

Inferred type
VV : {v : Int | v > ?a}
not a subtype of Required type
VV : {VV : Int | VV == x + 2}
In Context
x : Int
?a : {?a : Int | ?a > x}

add2 0 = 3
violating add2’s specification
if incr 0 = 2
To verify a caller, modular verifiers use callee’s specification.
MODULAR VERIFICATION

add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)

incr :: x:Int -> { y:Int | y = x + 1 }
incr x = x + 1

To verify a caller, modular verifiers use callee’s specification

add2 0 = 3
violating add2’s specification
if incr 0 = 2
OVERVIEW
OVERVIEW

Code

Users Specification

Synthesized Specification

Synthesizer

Verifier

Verified!

Concrete Counterexample (to users specification)

Concrete Counterexample (to synthesized specification)

Abstract Counterexample

G2

Synthesized Specification
OVERVIEW
COUNTEREXAMPLES

Concrete Counterexample

map :: (a -> b) -> xs:[a] -> { ys:[b] | size xs == size ys}
map f [] = []
map f (x:xs) = map f xs

Abstract Counterexample

add2 :: x:Int -> { y:Int | y == x + 2 }
add2 x = incr (incr x)
incr :: x:Int -> { y:Int | y > x }
incr x = x + 1

map id [1] = []
add2 0 = 3
if incr 0 = 2
OVERVIEW
EXAMPLE

concat :: x:([a]) -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> z:[a]
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys = x:app xs ys

Abstract counterexample:
concat [[], []] = [0]
if app [] [] = [0]

Real evaluation:
app [] [] = []

Synthesis
pre_app([], []) ⇒ ¬post_app([], [], [0])
pre_app([], []) ⇒ post_app([], [], [])
EXAMPLE

\begin{align*}
\text{concat} &:: \text{x}:[\text{a}] \to \{ \text{v} : [\text{a}] \mid \text{size v} = \text{sumsize x} \} \\
\text{concat} &\quad [ ] = [ ] \\
\text{concat} &\quad (\text{xs} : [ ]) = \text{xs} \\
\text{concat} &\quad (\text{xs} : (\text{ys} : \text{xss})) = \text{concat} \ ((\text{app} \ \text{xs} \ \text{ys}) : \text{xss}) \\
\text{app} &:: \text{x}:[\text{a}] \to \text{y}:[\text{a}] \to \{ \text{z}:[\text{a}] \mid \text{size z} = 0 \} \\
\text{app} &\quad [ ] \ [ ] = [ ] \\
\text{app} &\quad \text{xs} \ [ ] = \text{xs} \\
\text{app} &\quad [ ] \ \text{ys} = \text{ys} \\
\text{app} &\quad (\text{x} : \text{xs}) \ \text{ys} = \text{x} : \text{app} \ \text{xs} \ \text{ys}
\end{align*}

Abstract counterexample:
\[
\text{concat} \ [ ] \ [ ] = [0] \\
\text{if} \ \text{app} \ [ ] \ [ ] = [0]
\]

Real evaluation:
\[
\text{app} \ [ ] \ [ ] = [ ]
\]

Synthesis constraints:
\[
\text{pre}_{\text{app}}([], []) \Rightarrow \neg \text{post}_{\text{app}}([], [], [0]) \\
\text{pre}_{\text{app}}([], []) \Rightarrow \text{post}_{\text{app}}([], [], [])
\]
EXAMPLE

concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == 0 }
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys = x:app xs ys

Concrete counterexample:
app [0] [] = [0]

Synthesis constraints:
pre_app([], []) ⇒ ¬post_app([], [], [0])
pre_app([], []) ⇒ post_app([], [], [])
pre_app([0], []) ⇒ post_app([0], [], [0])
EXAMPLE

concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x }
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys = x:app xs ys

Concrete counterexample:
app [0] [] = [0]

Synthesis constraints:
pre_app([], []) ⇒ ¬post_app([], [], [0])
pre_app([], []) ⇒ post_app([], [], [])
pre_app([0], []) ⇒ post_app([0], [], [0])
EXAMPLE

concat :: x: [a] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x: [a] -> y: [a] -> {z: [a] | size z == size x}
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys = x:app xs ys

Concrete counterexample:
app [0] [0] = [0, 0]

Synthesis constraints:
pre_{app}([], []) ⇒ ¬post_{app}([], [], [0])
pre_{app}([], []) ⇒ post_{app}([], [], [])
pre_{app}([0], []) ⇒ post_{app}([0], [], [0])
pre_{app}([0], [0]) ⇒ post_{app}([0], [0], [0, 0])
EXAMPLE

concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x + size y}
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys = x:app xs ys

Concrete counterexample:
app [0] [0] = [0, 0]

Synthesis constraints:
pre_app([], []) ⇒ ¬post_app([], [], [0])
pre_app([], []) ⇒ post_app([], [], [])
pre_app([0], []) ⇒ post_app([0], [], [0])
pre_app([0], [0]) ⇒ post_app([0], [0], [0])
EXAMPLE

concat :: x:[[a]] -> {v : [a] | size v = sumsize x}
concat [] = []
concat (xs:[]) = xs
concat (xs:(ys:xss)) = concat ((app xs ys):xss)

app :: x:[a] -> y:[a] -> { z:[a] | size z == size x + size y}
app [] [] = []
app xs [] = xs
app [] ys = ys
app (x:xs) ys = x:app xs ys

Concrete counterexample:
app [0] [0] = [0, 0]

Synthesis constraints:
pre_app([], []) ⇒ ¬post_app([], [], [0])
pre_app([], []) ⇒ post_app([], [], [])
pre_app([0], []) ⇒ post_app([0], [], [0])
pre_app([0], [0]) ⇒ post_app([0], [0], [0])
Walk down the call graph, from level 1 to level k.

At level i, synthesize specifications for the functions at level i + 1 that would (if correct) prove specifications of functions at level i.

Backtrack if:
- a concrete counterexample to a specification at level <= i is found
- specification synthesis problem becomes unrealizable
OVERVIEW

Code

Users Specification

Specification

Verifier

Verified!

Concrete

Counterexample
(tosynthesized
specification)

Synthesized

Specification

Concrete

Counterexample
(touusers
specification)

LIA

Specification

Synthesizer

Abstract

Counterexample
(tosynthesized
specification)
SYNTHEZIZER

Constraints

Synthesize specification

Measures
(size, sumsize)

Realizable

Unrealizable

Verifier
Specification

Interpolant
SYNTESIZER

Convert to Integer template

Synthesize specification over LIA (using SMT)

Convert back to full features

Realizable

Unrealizable

Verifier Specification

Interpolant

Constraints

Measures (size, sumsize)

Measures (size, sumsize)
<table>
<thead>
<tr>
<th>Specification Type</th>
<th>Specification Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Literal</td>
<td><code>f :: \{ x : \text{Int} \mid x &lt; 0 \} \rightarrow \{ y : \text{Int} \mid y &gt; 0 \} \rightarrow \text{[Int]}</code></td>
</tr>
<tr>
<td>Inputs/Outputs</td>
<td><code>f x y = [x + 4, y + 4]</code></td>
</tr>
<tr>
<td>Integer Measures</td>
<td><code>f :: \text{Int} \rightarrow \{ \text{xs} : \text{[Int]} \mid \text{size xs} &gt; 0 \}</code></td>
</tr>
<tr>
<td></td>
<td><code>\text{size} :: \text{[a]} \rightarrow \text{Int}</code></td>
</tr>
<tr>
<td></td>
<td><code>\text{sumsize} :: \text{[[a]]} \rightarrow \text{Int}</code></td>
</tr>
<tr>
<td>ADT Contents</td>
<td><code>f :: \text{Int} \rightarrow \{ \{ \text{x} : \text{Int} \mid x &gt; 0 \} \}</code></td>
</tr>
</tbody>
</table>
Synthesize LIA specifications for: \( f : \text{Int} \to \text{Int} \to [\text{Int}] \)

\[ f \ x \ y = [x + 4, y + 4] \]

Constraint

\[ \text{pre}_f(0, 1) \Rightarrow \text{post}_f(0, 1, [4, 5]) \]

Integer Measures

\( \text{size} [4, 5] = 2 \)

\[ \text{pre}_f(0, 1) \Rightarrow \text{post}_f(0, 1, 2) \]

\[ \text{post}_f(x, y, z) = z > 0 \]

\[ \text{post}_f(x, y, z) = \{ z : a \mid \text{size} z > 0 \} \]
Synthesize LIA specifications for: \( f :: \text{Int} \to \text{Int} \to \text{[Int]} \)

\[ f \; x \; y = [x + 4, y + 4] \]

**Constraint**

\[ \text{pre}_f(0, 1) \Rightarrow \text{post}_f(0, 1, [4, 5]) \]

**ADT Contents**

\[ \text{post}_f(\text{cons})(x, y, r) \]

\[ \text{pre}_f(0, 1) \Rightarrow \text{post}_f(0, 1, 2) \]

\[ \land \; \text{post}_f(\text{cons})(0, 1, 4) \]

\[ \land \; \text{post}_f(\text{cons})(0, 1, 5) \]

\[ \text{post}_f(x, y, r) = r > 0 \]

\[ \text{post}_f(\text{cons})(x, y, r) = r > 2 \]

\[ \{ r : \{ x : \text{Int} \mid x > 2 \} \mid \text{size} \; r > 0 \} \]
SOUNDNESS AND COMPLETENESS

**Soundness Theorem** –
Assuming a sound verifier, counterexample generator, and synthesizer, the inference algorithm is sound.

**Completeness Definition** –
We say an inference function is complete if, whenever there exists some set of specifications that will allow verification, the algorithm succeeds in finding such a set.

**Completeness Theorem** –
Assuming a finite number of possible specifications, and a sound and complete verifier, counterexample generator, and synthesizer, the inference algorithm is complete.
EVALUATION

Ran the inference algorithm on 15 benchmarks, some created by us, some drawn from a graduate student level class homework assignment.

Largest benchmark is the inner loop of a kmeans implementation, involving 34 functions. We prove the codes specifications in 596 seconds (slightly under 10 minutes.)
SUMMARY

• For verification to succeed, modular verifiers require specifications to not only be correct, but be sufficiently supported by callee’s specifications.

• Given specifications written by the user, our inference algorithm automatically finds the required set of specifications for a modular verifier to succeed.

• Using an SMT solver to synthesizer LIA specifications allows us SyGuS like synthesis, but to also prove unrealizability and get interpolants.

• Our approach is implemented to find LiquidHaskell specifications, using G2 as a counterexamples generator, and it’s effectiveness is demonstrated on a variety of benchmarks.