Syntax Guided Synthesis
Day 1

Armando Solar-Lezama
Synthesis: ExCape view

\[ R = \{ p_0 \ldots p_i \} \]

\[ \varphi(p) = \forall \text{in. } \ldots \ p(\text{in}) \ldots \]
Syntax-Guided Program Synthesis

Common theme to many recent efforts
- Sketch (Bodik, Solar-Lezama et al)
- FlashFill (Gulwani et al)
- Implicit programming: Scala^Z3 (Kuncak et al)
- Super-optimization (Schkufza et al)
- Invariant generation (Many recent efforts...)
- TRANSIT for protocol synthesis (Udupa et al)
- Oracle-guided program synthesis (Jha et al)
- Auto-grader (Singh et al)
Key questions

- How do you define a space of programs?
- How do you search it efficiently?
- How do you know when you have found the right answer?
Tools

SyGuS
- General formalism for expressing synthesis problems
- Pros
  - Community effort
  - Multiple implementations based on different algorithms
  - Clean formalism
- Cons
  - Limited expressiveness

Sketch
- Synthesis enabled language
- Pros
  - Full featured language
- Cons
  - Single implementation based on one algorithm
  - SyGuS frontend for sketch
SyGuS - example

Theory QF-LIA

- Types: Integers and Booleans
- Logical connectives, Conditionals, and Linear arithmetic
- Quantifier-free formulas

Function to be synthesized \( f(\text{int } x, \text{int } y) : \text{int} \)

Specification: \( x \leq f(x, y) \land y \leq f(x, y) \land (f(x, y) = x \lor f(x, y) = y) \)

Candidate Implementations: Linear expressions

\[
\text{LinExp} := x \mid y \mid \text{Const} \mid \text{LinExp} + \text{LinExp} \mid \text{LinExp} - \text{LinExp}
\]

No solution exists
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 (>= (max2 x y) x))
(constraint (>= (max2 x y) y))
(constraint (or (= x (max2 x y)) (= y (max2 x y))))
(check-synth)
You want to partition $N$ elements over $P$ procs

- How many elements should a processor get?

Obvious answer is $N/P$

Obvious answer is wrong!
void partition(int p, int P, int N, ref int ibeg, ref int iend) {

    N * P + N\%P

}
Tests as specifications

How does the system know what a partition is?

```c
harness void testPartition(int p, int N, int P){
    if (p>=P || P < 1){
        return;
    }
    int ibeg, iend;
    partition(p, P, N, ibeg, iend);
    assert iend - ibeg < (N/P) + 2;
    if (p+1 < P){
        int ibeg2, iend2;
        partition(p+1, P, N, ibeg2, iend2);
        assert iend == ibeg2;
    }
    if (p==0){
        assert ibeg == 0; }
    if (p==P-1){
        assert iend == N; }
}
```
Overview

Describing program spaces

Counterexample guided synthesis

Synthesis as search: 3 approaches

The limits of CEGIS
Syntax Guided Synthesis

Space of programs is defined syntactically

Structural hypothesis

• What is the space
  • How do you describe it (user’s perspective)
  • How do you represent it (system’s perspective)

• Does it have any properties that can help the search
Example

\[ 1,4,2,0,7,9,2,5,0,3,2,4,7 \rightarrow 1,2,4,0,2,5,7,9,0,2,3,4,7,0 \]

Process(in) := \( \text{sort(lstExpr[0, firstZero(in)]) + [0] + recursive(lstExpr[firstZero(in)+1, len(in)])} \);
What is the space?

\[ \text{lstExpr} := \text{sort(lstExpr)} \]
\[ \text{lstExpr}[\text{intExpr},\text{intExpr}] \]
\[ \text{lstExpr} + \text{lstExpr} \]
\[ \text{recursive(lstExpr)} \]
\[ [0] \]
\[ \text{in} \]
\[ \text{intExpr} := \text{firstZero(lstExpr)} \]
\[ \text{len(lstExpr)} \]
\[ 0 \]
\[ \text{intExpr} + 1 \]

The set of all programs in lstExpr
What is the space?

Grammars as definitions of program spaces

• Pro
  • Clean declarative description
  • Easy to sample
  • Easy to explore exhaustively

• Con
  • Insufficiently expressive
What if we know the following:

- Sort is never called more than once in a sub-list.
- Recursive calls should be made on lists whose length is less than $\text{len}(\text{in})$.
- We’ll never have to add one multiple times in a row.
Grammars in SyGuS

(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))))
Generators/Generative models

Programs that produce programs
  • Can be either random or non-deterministic

Pros:
  • Extremely general
    • easy to enforce arbitrary constraints

Cons:
  • Extremely general
    • Hard to analyze and reason about
    • Hard to automatically discover structure of the space
Symmetries

Multiple ways of representing the same problem

\[ \text{Expr} := \text{var}*\text{const} \]
\[ \quad | \ \text{Expr} + \text{Expr} \]
\[ w*\text{c1}+(x*\text{c2}+(y*\text{c3}+z*\text{c4})) \]

- Grammar on the right has fewer symmetries
- Grammar on the left can produce all possible ways to parenthesize
- Can completely eliminate symmetries from the right by enforcing a variable ordering
  - Can’t be done with a grammar, but it can with a generative model

\[ \text{Expr(}v\text{min}) := \text{let } v = \text{var}() \text{ in } v*\text{const} \ (\text{assert } v > v\text{min}) \]
\[ \quad | \ \text{let } v=\text{var}() \text{ in } v*\text{const} + \text{Expr}(v) \ (\text{assert } v > v\text{min}) \]
Symmetries

Do symmetries matter?
• It depends

Some methods are very sensitive to symmetries
• E.g. symbolic search

Others are largely oblivious to them
• E.g. sampling
CEGIS
The general synthesis problem

\[ \exists P \forall in \ (in, P \models Spec) \]
Ensuring correctness

This is a hard problem in general

\[ \forall \text{ in } (\text{in}, P \models Spec) \]

Two points of view:

• Not my problem
  • This is not the verification summer school

• Synthesis can make verification simpler
  • Synthesize code that is easier to prove correct
Counterexample guided inductive synthesis

Ideas

• Rely on an oracle to tell you if your program is correct
• If it is not, rely on oracle to generate counterexample inputs
• Reduce to an inductive synthesis problem
CEGIS

Synthesize

\[ \exists P \text{ s.t. } \text{Correct}(P, \text{in}_i) \]

\{\text{in}_i\}

Check

\[ \exists \text{in} \text{ s.t. } \neg \text{Correct}(P, \text{in}_i) \]

Insert your favorite checker here
∃ a \forall b \exists c \forall d \exists e \forall f \exists g \forall h \exists i \forall j \exists k (a, b, c, d, e, f, g, h, i, j, k)
CEGIS in Sketch

\[ Q(c, in) \]

**Synthesize**

\[ Q(c, in_0) \]
\[ Q(c, in_2) \]
\[ Q(c, in_3) \]

**Check**

\[ \neg Q(c, in_2) \]

\[ \neg Q(c, in_3) \]
Syntax Guided Synthesis
Day 2

Armando Solar-Lezama
Explicit Search
Idea

Generate programs one by one
  • Generate & Test approach

Key issues
  • In what order do you generate?
    • Influences performance *and* result quality
  • How do you prune?
    • Essential for scalability
  • How do you keep track of the remaining space?
    • Especially challenging in the context of pruning
Explicit search from grammars

Grammar describes how to generate program fragments from smaller program fragments

\[ \text{plist} := \text{set of all terminals} \]
\[ \text{while (true)} \{ \]
\[ \quad \text{plist} := \text{grow(plist)}; \]
\[ \quad \text{forall ( p in plist)} \]
\[ \quad \quad \text{if(isCorrect(p))} \{ \text{return p; } \} \]
\[ \} \]
\[ \text{grow(plist)} \{ \]
\[ \quad \text{// return a list of all trees generated by} \]
\[ \quad \text{// taking a non-terminal and adding} \]
\[ \quad \text{// nodes in plist as children} \]
\[ \} \]
Explicit search from grammars

Grammar describes how to generate program fragments from smaller program fragments

Level 0

<table>
<thead>
<tr>
<th>in</th>
<th>[0]</th>
<th>0</th>
</tr>
</thead>
</table>

Level 1

<table>
<thead>
<tr>
<th>in</th>
<th>[0][0]</th>
<th>in + in</th>
<th>in + [0]</th>
<th>[0] + [0]</th>
<th>[0] + in</th>
<th>rec(in)</th>
</tr>
</thead>
<tbody>
<tr>
<td>rec([0])</td>
<td>firstZero(in)</td>
<td>firstZero([0])</td>
<td>len(in)</td>
<td>len([0])</td>
<td>0+1</td>
<td></td>
</tr>
</tbody>
</table>

Set grows very fast!

Large equivalence classes of equivalent programs
Identifying equivalent programs

Program equivalence is hard
  • It is also unnecessary!

Observational Equivalence
  • Are they equivalent wrt the inputs
    • easy to check efficiently
    • sufficient for the purpose of PBE
  • Keep only the simplest one

plist := set of all terminals
while(true){
    plist := grow(plist);
    plist := elimEquivalents(plist);
    forall( p in plist)
      if(isCorrect(p)){ return p; }
}
Explicit search from grammars

Features:

• Search small programs before large programs
• Simple
• Works even with black-box language building blocks
  • no need to have source for sort or firstZero just need to be able to execute them
  • no need to know of any properties about them e.g. automatically ignores sort(sort(in)) without having to know that sort is idempotent
• Complexity depends on the size of the set of distinct programs
  • Copes well with symmetries
Explicit search from grammars

Limitations:

• Only scales to very small programs
• Unsuitable for programs with unknown constants
  • A single unknown 32-bit constant makes the problem intractable
• Hard to generalize to arbitrary generators
  • Relies heavily on recursive structure of grammar
• Hard to take advantage of additional domain knowledge

Example systems:

• Transit [Udupa et al. PLDI 2013]
• Recursive Program Synthesis [Albarghouthi et al., CAV 2013]
Symbolic Search
The general synthesis problem

\[ \exists a \forall m \in \text{in}(\text{in}(\text{in}(kPc) \equiv \text{Spec}c)) \]
Example: Least Significant Zero Bit

- 0010 0101 $\rightarrow$ 0000 0010

```c
int W = 32;

bit[W] isolate0 (bit[W] x) {
    bit[W] ret = 0;
    for (int i = 0; i < W; i++)
        if (!x[i]) { ret[i] = 1; return ret; }
}
```

Trick:
- Adding 1 to a string of ones turns the next zero to a 1
- i.e. 000111 + 1 = 001000
Sample Generator

/**
 * Generate the set of all bit-vector expressions involving +, &, xor and bitwise negation (~).
 * the bnd param limits the size of the generated expression.
 */

generator bit[W] gen(bit[W] x, int bnd){
    assert bnd > 0;
    if(??) return x;
    if(??) return ??;
    if(??) return ~gen(x, bnd-1);
    if(??){
        return { | gen(x, bnd-1) (+ | & | ^) gen(x, bnd-1) |};
    }
}
∃ c ∀ \( \forall i \in Q \), \( \forall i \in Q \), \( c \subseteq \text{Spec} \)
A sketch as a constraint system

```c
int lin(int x) {
    if (x > 1)
        return 2*x + 3;
    else
        return 4*x;
}

void main(int x) {
    int t1 = lin(x);
    int t2 = lin(x+1);
    if (x<4) assert t1 >= x*x;
    if (x>=3) assert t2 - t1 == 1;
}
```
Ex: Population count.

```c
int pop (bit[W] x) {
    int count = 0;
    for (int i = 0; i < W; i++) {
        if (x[i]) count++;
    }
    return count;
}
```

\[ F(x) = \]
int popSketched (bit[W] x) implements pop {
    repeat(??) {
        x = (x & ??) + ((x >> ??) & ??);
    }
    return x;
}
Performance considerations

Formula size is the biggest challenge

Space of possible programs matters a lot too
MCMC Probabilistic Search

Based on “The Markov Chain Monte Carlo Revolution”
Persi Diaconis
Markov Chains

Let $\mathcal{X}$ be a finite set

A Markov chain is defined by a matrix $K(x, y): \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$

- $K(x, y) \geq 0$
- $\sum_y K(x, y) = 1$

Probability of a series $X_0, X_1, X_2 \ldots$

- $P(X_1 = y | X_0 = x) = K(x, y)$
- $P(X_1 = y, X_2 = z | X_0 = x) = K(x, y)K(y, z)$
- $P(X_2 = z | X_0 = x) = \sum_y K(x, y)K(y, z)$
  - This is matrix multiplication!
Stationary distribution

What is the probability $\pi(x)$ of being in a node x at some arbitrary step?

- $\pi(x) > 0$ and $\sum_x \pi(x) = 1$
- $\pi(y) = \sum_x \pi(x)K(x, y)$
  - i.e. $\pi = \pi K$
Fundamental theorem of (finite) Markov chains

If there is an $n_0$ s.t. $\forall x, y. \ n > n_0 \Rightarrow K^n(x, y) \geq 0$
- i.e. the matrix is connected.

$\forall x. \ \lim_{n \to \infty} K^n(x, y) = \pi(y)$
- The n’th step of a run starting at $x$ has probability close to $\pi(y)$ of being at $y$ if $n$ is large.
MCMC Based synthesis

Approach:

• Let $\chi$ be the space of programs
• Engineer a $K(x, y)$ such that $\pi(x)$ is high for “good programs” and low for “bad programs”
• Pick a random start state $x_0$
• Simulate the markov process for $n$ steps for some large $n$.
• By the fundamental theorem, the probability that $x_n$ is a good program will be higher than the probability that it is a bad program
Metropolis algorithm

Start with a markov matrix $J(x, y)$ with $J(x, y) > 0 \leftrightarrow J(y, x) > 0$

$$K(x, y) = \begin{cases} J(x, y) & \text{if } x \neq y, \ A(x, y) \geq 1 \\ J(x, y)A(x, y) & \text{if } x \neq y, \ A(x, y) < 1 \\ J(x, y) + \sum_{z:A(x,z)<1} J(x, z)(1 - A(x, z)) & \text{if } x = y \end{cases}$$

$A(x, y)$ is the acceptance ratio $\frac{\pi(y)J(y,x)}{\pi(x)J(x,y)}$

Note $\pi(x)K(x, y) = \pi(y)K(y, x)$

- Then $\sum_x \pi(x)K(x, y) = \sum_x \pi(y)K(y, x) = \pi(y) \sum_x K(y, x) = \pi(y)$
Tradeoffs

Symbolic
• Pros
  • Very good at discovering unknown constants
  • Flexible
  • Good for large spaces with simple components
• Cons
  • Must be able to reason symbolically about the entire program
  • Complexity of the spec affects complexity of synthesis
  • Requires significant engineering of program space

Enumerative
• Pros
  • Very good at ignoring symmetries
  • Complexity is independent of complexity of spec or components
• Cons
  • Only scales to small programs
  • Fails in the context of unknown constants

Stochastic
• Pros
  • Complexity is independent of complexity of spec
  • Easy to incorporate quantitative criteria
  • Can discover large programs given a good proposal distribution
• Cons
  • Bad for “needle in a haystack” problems
  • Proposal distribution may require significant engineering