Lecture #10

Syntax
Guided
Synthesis

Software Synthesis and Automated Reasoning
Reasoning

λ-calculus

Dependent Types

Type Theory (basics)

Axiomatic Semantics

Satisfiability Modulo Theory

Programming by Example

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Type Directed Synthesis

Refinement Types

Synthesis
Synthesis!

MIT & NASA, 1957

"Code" (≈165cm)
Dimensions in program synthesis

[Gulwani 2010]

Behavioral constraints
how do you tell the system what the program should do?

Structural constraints
what is the space of programs to explore?

Search strategy
How does the system find the program you want?

By Example
(previous lecture)
Syntax-Guided Synthesis (= SyGuS)

\[ [1,4,7,2,0,6,9,2,5,0,3,2,4,7] \rightarrow [1,2,4,7,0] \]

\[ f(x) := \text{sort}(x[0..\text{find}(x, 0)]) + [0] \]

\[ L ::= \text{sort}(L) \mid L[N..N] \mid L + L \mid [N] \mid x \]

\[ N ::= \text{find}(L,N) \mid 0 \]
• Context-free grammar

L ::= sort(L)    
   L[N..N]    
   L + L      
   [N]        
   x

N ::= find(L,N)  
   0

CFG FTW
CFGs as Structural Constraints

Space of programs

= all complete programs generated by rewriting the starting nonterminal according to productions

\[ L ::= \text{sort}(L) \]
\[ L[N..N] \]
\[ L + L \]
\[ [N] \]
\[ x \]

\[ N ::= \text{find}(L,N) \]
\[ 0 \]

\[ x \quad \text{sort}(x) \quad x + x \quad x[0..0] \]

\[ \ldots \]

\[ x[0..\text{find}(x, 0)] \]

\[ \ldots \]

\[ \text{sort}(x[0..\text{find}(x, 0)]) + [0] \]

\[ \ldots \]
The SyGuS Project

SyGuS problem = \langle \text{theory, spec, grammar} \rangle

A “library” of types and function symbols

Example: Linear Integer Arithmetic (LIA)

True, False

0, 1, 2, ...

\&, \lor, \neg, +, \leq, \text{ite}

CFG with terminals in the theory (+ input variables)

Example: Conditional LIA expressions w/o sums

E ::= x | \text{ite} C E E

C ::= E \leq E | C \& C | \neg C
The SyGuS Project

SyGuS problem = \langle \text{theory, spec, grammar} \rangle

A first-order logic formula over the theory

By Example:

\begin{align*}
f(0, 1) &= 1 \\ f(1, 0) &= 1 \\ f(1, 2) &= 2 \\ f(3, 0) &= 3
\end{align*}

With free variables:

\begin{align*}
x &\leq f(x, y) \\ y &\leq f(x, y) \\ (f(x, y) = x \vee f(x, y) = y)
\end{align*}
Dimensions in program synthesis

By Example (previous lecture)

Behavioral constraints
how do you tell the system what the program should do?

Structural constraints
what is the space of programs to explore?

Search strategy
How does the system find the program you want?

Context-Free Grammar

[Gulwani 2010]
Enumerative Search

=  Explicit / Exhaustive Search

**Idea:** Generate programs from the grammar, one by one, and test them on the examples

(well, duh)
How big is the space?

\[ E ::= x \mid E @ E \]

**depth \leq 1**

\[ N(1) = 1 \]

**depth \leq 2**

\[ N(2) = 2 \]

**depth \leq 3**

\[ N(3) = 5 \]

\[ N(d) = 1 + N(d - 1)^2 \]
How big is the space?

\[ E ::= x \mid E \oplus E \]

\[ N(d) = 1 + N(d - 1)^2 \quad \text{N(d)} \sim c^{2^d} \]

\[ \begin{align*}
N(1) &= 1 \\
N(2) &= 2 \\
N(3) &= 5 \\
N(4) &= 26 \\
N(5) &= 677 \\
N(6) &= 458330 \\
N(7) &= 210066388901 \\
N(8) &= 44127887745906175987802 \\
N(9) &= 1947270476915296449559703445493848930452791205 \\
N(10) &= 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026 \\
\end{align*} \]
How big is the space?

\[
\begin{align*}
E & ::= x_1 \mid \ldots \mid x_k \\
& \quad \mid E @_1 E \mid \ldots \mid E @_m E
\end{align*}
\]

\[N(d) = k + m \cdot N(d - 1)^2\]

\[
\begin{align*}
N(1) &= 3 \\
N(2) &= 30 \\
N(3) &= 2703 \\
N(4) &= 21918630 \\
N(5) &= 144127902320703 \\
N(6) &= 6231855668414547953818685622630 \\
N(7) &= 116508075215851596766492219468227024724121520304443212304350703 \\
N(8) &= \cdots
\end{align*}
\]

Answer: Pretty darn big.
Top-down Enumeration

- Start from initial symbol
- Repeatedly expand nonterminals using productions

L ::= sort(L) | L[N..N] | L + L | [N] | x
N ::= find(L,N) | 0

{[1,4,0,6] → [1,4]}

nonterminals  rules (productions)

start from initial symbol

spec

top-down (⟨T, V, R, S⟩, {x → y})

P := [S]

while (P ≠ [])
    p = P.dequeue();
    if (ground(p) ∧ p([x]) = [y])
        return p;
    P.enqueue(expand(p))

expand(p, R)

L → [sort(L), L[N..N], L + L, [N], x]
L[N..N] → [sort(L)[N..N], [N][N..N],
            L[0..N], L[find(L,N)..N], L[N..0], ]
Bottom-up Enumeration

- Start from terminals
- Combine sub-programs into larger programs using productions

\[
P := [V \sim t \mid V ::= t \in R \land \text{ground}(t)]
\]

\[
\text{while (true)} \quad P += \text{grow}(P);
\]

\[
\text{forall (p in P)} \quad \text{if (S \sim p \land p([x]) = [y])} \quad \text{return } p;
\]

**Terms and Non-terminals**

- **Terminals**
- **Non-terminals**
- **Rules (productions)**
- **Starting non-terminal**

**Bottom-up (\(\langle T, V, R, S \rangle, \{x \rightarrow y\}\))**

**Equations**

\[
L ::= \text{sort}(L) \mid L[N..N] \mid L + L \mid [N] \mid x
\]

\[
N ::= \text{find}(L,N) \mid 0
\]

\[
\{[1,4,0,6] \rightarrow [1,4]\}
\]
Bottom-up Enumeration

\[
L ::= \text{sort}(L) \\
L[N..N] \\
L + L \\
[N] \\
x \\
N ::= \text{find}(L,N) \\
0
\]

\[
\begin{align*}
\text{sort}(x) & \quad x[0..0] & \quad x + x & \quad [0] & \quad \text{find}(x,0) \\
\text{sort}(\text{sort}(x)) & \quad \text{sort}(x + x) & \quad \text{sort}(x[0..0]) \\
\text{sort}([0]) & \quad x[0..\text{find}(x,0)] & \quad x[\text{find}(x,0)..0] \\
x[\text{find}(x,0)..\text{find}(x,0)] & \quad \text{sort}(x)[0..0] \\
x[0..0][0..0] & \quad (x + x)[0..0] & \quad [0][0..0] \\
x + (x + x) & \quad x + [0] & \quad \text{sort}(x) + x \\
x[0..0] + x & \quad (x + x) + x & \quad [0] + x \\
x + x[0..0] & \quad x + \text{sort}(x) \\
\ldots
\end{align*}
\]
How to make it scale

Pruning
Discard useless subprograms

\[ m \cdot N^2 \quad \text{and} \quad m \cdot (N-1)^2 \]

Ranking
Explore more promising candidates first

\[ P = \{ [0][N..N], x[N..N], \ldots \} \]

dqueue this first
Pruning

• When can we discard a subprogram?

- It’s equivalent to something we have already explored
- No matter what we combine it with, it cannot satisfy the spec

Equivalence reduction
(also: symmetry breaking)

Top-down propagation

\[
\begin{align*}
\text{sort}(x) & \quad \rightarrow \quad \text{sort}([N] + L) \\
\text{sort}(\text{sort}(x)) & \quad \rightarrow \quad [N] + L
\end{align*}
\]
Equivalent Subprograms

\[
\begin{align*}
L & ::= \text{sort}(L) \\
 & \quad \text{L}[N..N] \\
 & \quad L + L \\
 & \quad [N] \\
 & \quad x \\
N & ::= \text{find}(L,N) \\
 & \quad 0
\end{align*}
\]

\[
\begin{align*}
sort(x) & \  x[0..0] \  x + x \  [0] \  \text{find}(x,0) \\
sort(\text{sort}(x)) & \  sort(x + x) \  sort(x[0..0]) \\
sort([0]) & \  x[0..\text{find}(x,0)] \  x[\text{find}(x,0)..0] \\
x[\text{find}(x,0)..\text{find}(x,0)] & \  \text{sort}(x)[0..0] \\
x[0..0][0..0] & \  (x + x)[0..0] \  [0][0..0] \\
x + (x + x) & \  x + [0] \  \text{sort}(x) + x \\
x[0..0] + x & \  (x + x) + x \  [0] + x \\
x + x[0..0] & \  x + \text{sort}(x)
\end{align*}
\]

bottom-up
Equivalent Subprograms

L ::= sort(L)
L[N..N]
L + L
[N]
x
N ::= find(L,N)
0

bottom-up

x 0

sort(x) x[0..0] x + x [0] find(x,0)

sort(sort(x)) sort(x + x) sort(x[0..0])
sort([0]) x[0..find(x,0)] x[find(x,0)..0]
x[find(x,0)..find(x,0)] sort(x)[0..0]
x[0..0][0..0] (x + x)[0..0] [0][0..0]
x + (x + x) x + [0] sort(x) + x
x[0..0] + x (x + x) + x [0] + x
x + x[0..0] x + sort(x)

...
Equivalent Subprograms

\[
L ::= \text{sort}(L) \\
   L[N..N] \\
   L + L \\
   [N] \\
   x \\
N ::= \text{find}(L,N) \\
   0
\]

\[
\begin{align*}
x + (x + x) & \quad x + [0] \\
\text{sort}(x) + x \quad [0] + x & \quad \text{find}(x,0)
\end{align*}
\]
Alright then

- We’ll just remove all programs equivalent to $p$ from $P$ and be done with it.
  - In general: *undecidable*. 
  - In the case of SyGuS theories: decidable but slow (NP-hard)
  - Performing expensive checks for every candidate defeats the purpose of pruning the space!
Observational Equivalence

• In PBE, all we care about is equivalence on the given inputs!
  ‣ easy to check efficiently
  ‣ even more programs are equivalent...

\[ p_1 =_o p_2 \iff p_1([i]) = p_2([i]) \]
Observational Equivalence

\{ [0] \rightarrow [0] \} 

\begin{align*}
\text{sort}(x) & \quad \text{x[0..0]} \quad \text{x + x} \\
\text{sort}(x + x) & \quad \text{x[0..\text{find}(x,0)]}
\end{align*}

\begin{align*}
\text{x + (x + x)} & \quad \text{x + [0]} \\
\text{sort}(x) + x & \quad [0] + x \\
\text{x + sort(x)}
\end{align*}

\ldots
Observational Equivalence

\{[0] \rightarrow [0]\}

\begin{align*}
  x & \rightarrow 0 \\
  \text{sort}(x) & \rightarrow x[0..0] \\
  x + x & \rightarrow x + \text{find}(x,0) \\
  \text{sort}(x + x) & \\
  x[0..\text{find}(x,0)] & \\
  x + (x + x) & \\
  x + [0] & \rightarrow \text{sort}(x) + x \\
  [0] + x & \\
  x + \text{sort}(x) & \ldots
\end{align*}
Observational Equivalence

\[
\{[0] \rightarrow [0]\}
\]

\[
x \rightarrow 0
\]

\[
x[0..0] x + x
\]

Prune early = Prune BIG

\[
x + (x + x)
\]

\[
\ldots
\]

Used in almost all PBE tools:

- ESolver [Udupa et al. ’13]
- Escher [Albarghouthi et al. ’13]
- Lens [Phothilimthana et al. ’16]
- EUSolver [Alur et al. ‘17]

...
Built-in Equivalences

- For a predefined vocabulary, equivalence reduction can be hard-coded in the tool or built into the grammar.

\[
x + y + z = x + (y + z)
\]
Built-in Equivalences

• For a predefined vocabulary, equivalence reduction can be hard-coded in the tool or built into the grammar.

\[
\begin{align*}
\text{L} &::= \text{sort}(\text{L}) \\
&\quad | \quad \text{L}[\text{N}..\text{N}] \\
&\quad | \quad \text{L} + \text{L} \\
&\quad | \quad [\text{N}] \\
&\quad | \quad \text{x} \\
\text{N} &::= \text{find}(\text{L}, \text{N}) \\
&\quad | \quad 0
\end{align*}
\]

\[
\begin{align*}
\text{L} &::= \text{L}_1 \\
&\quad | \quad \text{L}_1 + \text{L} \\
\text{L}_1 &::= \text{L}_2 \\
&\quad | \quad \text{sort}(\text{L}_3) \\
\text{L}_2 &::= \text{L}[\text{N}..\text{N}] \\
&\quad | \quad [\text{N}] \\
&\quad | \quad \text{x} \\
\text{L}_3 &::= \text{L}_2 \\
&\quad | \quad \text{L}_1 + \text{L} \\
\text{N} &::= \text{find}(\text{L}, \text{N}) \\
&\quad | \quad 0
\end{align*}
\]

\[\text{sort(sort}(\text{L})) = \text{sort}(\text{L})\]

Used by Leon [Kneuss et al.'13], \(\lambda^2\) [Feser et al.'15], ...
User-specified Equivalences

• Input:

  sort(sort(l)) = sort(l)
  (l + l) + l = l + (l + l)
  n = n + 0
  n + m = m + n

Equivalences

Knuth-Bendix

1. sort(sort(l)) → sort(l)
2. (l + l) + l → l + (l + l)
3. n + 0 → n
4. n + m →_{(n > m)} m + n

Term Rewriting System (TRS)

\[
\begin{align*}
  x & \rightarrow 0 \\
  \text{sort}(x) & \rightarrow x[0..0] x + x[0] \\
  \text{find}(x, 0) & \\
  \text{sort(sort(x))} & \rightarrow \text{normal}
\end{align*}
\]
Dimensions in program synthesis

[Gulwani 2010]

Behavioral constraints
how do you tell the system
what the program should do?

Structural constraints
what is the space of programs
to explore?

Context-Free Grammar

Programming

By Example

Rich Functional
Specifications

Enumerative

Search strategy
How does the system find
the program you want?
Why go beyond examples?

• Might need too many
  • *E.g.*: Myth needs 12 for `insert_sorted`, 24 for `list_n_th`
    ✦ Examples contain *too little* information
      ◦ (Successful tools use domain-specific ranking, *e.g.* FlashFill)

• Output difficult to construct
  • *E.g.*: AES cipher, red-black tree — involve large numbers/structures
    ✦ Examples also contain *too much* information (concrete outputs)

• Need strong guarantees
  • *E.g.*: one-way hash; aircraft controller

• Reasoning about non-functional properties
  • *E.g.*: time/space complexity, security, privacy
Why is this hard?

\[ \text{gcd} \ (\text{int} \ a, \ \text{int} \ b) \ \text{returns} \ (\text{int} \ c) \]

\text{requires} \ a > 0 \land b > 0

\text{ensures} \ c | a \land c | b

\forall d. (d | a \land d | b) \rightarrow c \leq d

\{
    \text{int} \ x , \ y := a, \ b;
    \text{while} \ (x \neq y) \{
        \text{if} \ (x > y) \ x := \cdots ;
        \text{else} \ y := \cdots ;
    \}
\}
Why is this hard?

Synthesis from examples

validation is easy!

Synthesis from specifications

validation is hard!
(and search is even harder)
Specifications in Logic

• It can be seen as a generalization of examples:

\[ \exists C. \forall x, y. \varphi(C, x, y) \Rightarrow \psi(x, y) \]

\[ \exists C. \bigwedge_i \varphi(C, x_i, y_i) \]

\[ \langle C, \sigma_x \rangle \rightarrow^* \langle \text{skip}, \sigma_y \rangle \]

\[ \varphi(C, x, y) \quad \text{spec} \]

\[ \psi(x, y) \]

\[ x = x_1 \rightarrow y = y_1 \]
\[ \land x = x_2 \rightarrow y = y_2 \]
\[ \land x = x_n \rightarrow y = y_n \]
Specifications in Logic

• We can use any logic we like
   (like we did for assertions in Hoare logic)

\[ \varphi(C, x, y) \equiv \langle C, \sigma_x \rangle \rightarrow^* \langle \text{skip}, \sigma_y \rangle \]

\[ \exists C. \ \forall x, y. \ \varphi(C, x, y) \Rightarrow \psi(x, y) \]

\[ x + 1 = y - x \]

\[ \psi(x, y) \] spec

program space

semantic encoding
Synthesis = ∃∀

One Program

Quantifier alternation
Notoriously hard for solvers

To Rule Them All

Hard to define in the presence of loops / recursion

∃C. ∀i, o. φ(C, i, o) ⇒ ψ(i, o)
Synthesis = \exists \forall

• Let’s try a simple program sketch.

\exists C. \forall x, y. \varphi(C, x, y) \Rightarrow \psi(x, y)

```
harness void main(int x) {
  int y := ?? * x + ??;
  assert x + 1 == y - x;
}
```

encoding

```
\exists c_1, c_2. \forall x, y. 
  y = c_1 \cdot x + c_2 \Rightarrow 
  x + 1 = y - x
```

simplify

```
\exists c_1, c_2. \forall x. 
  c_1 \cdot x + c_2 - 1 = x + x
```

How do we solve this constraint?
Counterexample-Guided Inductive Synthesis (CEGIS)

• **Idea 1**: Bounded Observation Hypothesis
  
  Assume that there exists a small set

  \[ X = \{ \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n \} \]

  such that

  \[ \bigwedge_{i} Q(C, \bar{x}_i) \implies \forall \bar{x}. Q(C, \bar{x}) \]

  Looks familiar?
Bounded Observation Example

- This is a linear constraint.
  - Two inputs are enough;
  - Furthermore: any two inputs!

\[ X = \{0, 1\} \]

\[ \exists c_1, c_2. \forall x. \ c_1 \cdot x + c_2 - 1 = x + x \]

\[ Q(c_1, c_2, 0) \equiv c_2 - 1 = 0 \]
\[ Q(c_1, c_2, 1) \equiv c_1 + c_2 - 1 = 2 \]

\[ Q(2, 1, x) \]

\[ c_1 = 2, \ c_2 = 1 \]

How do we find \( X \) in general?
CEGIS Loop

$$Q(\square, \square, x)$$

- **Solver** (for satisfiability of $\exists$-formulas)
- **Verifier** (for validity of $\forall$-formulas)

**Counterexample**

- $\exists c_1, c_2. c_1 \cdot 0 + c_2 - 1 = 0 + 0$
- $\land c_1 \cdot 1 + c_2 - 1 = 1 + 1$

**Bag of instances** ($X$)

- $x_1 = 0$
- $x_2 = 1$

**Hole assignments**

- $\forall x. 2 \cdot x + 1 - 1 = x + x$

**Verifier**

- $\exists c_1, c_2. \forall x. c_1 \cdot x + c_2 - 1 = x + x$
• We want to synthesize regular expressions

\[
\begin{align*}
\text{a} & \rightarrow 1 \\
\text{ab} & \rightarrow 1 \\
\text{ba} & \rightarrow 0
\end{align*}
\]

\[
\text{P} \rightarrow s = 0; \\
\quad \text{while } (\neg \text{eoi}) \{ \\
\quad \quad c = \text{next-letter}(); S; \\
\quad \} \\
\text{S} \rightarrow \text{if } s = Q \land c = C \text{ then } s := Q \\
\quad \quad \text{else } S \\
\text{Q} \rightarrow \cdots \text{states} \cdots \\
\text{C} \rightarrow \cdots \text{letters} \cdots 
\]

function sketch