Syntax

Guided

Synthesis

Lecture #10

Syntax
Guided
Synthesis
Reasoning

- $\lambda$-calculus
- Dependent Types
- Type Theory (basics)
- Axiomatic Semantics
- Satisfiability Modulo Theory

Synthesis

- Programming by Example
- Syntax Guided Synthesis
- Counterexample Guided Inductive Synthesis
- Type Directed Synthesis
- Refinement Types

AxiomaFc

SemanFcs

SaFsfiability

Modulo Theory

Programming by Example

Syntax Guided Synthesis

Counterexample Guided Inductive Synthesis

Type Directed Synthesis

Refinement Types
Synthesis!

MIT & NASA, 1957

“Code” (≈165cm)
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?
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

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

starting nonterminal

terminals

nonterminals

productions
CFGs as Structural Constraints

Space of programs

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

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

N ::= find(L,N)  
0

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

...  

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

...

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

...
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, ...
∧, ∨, ¬, +, ≤, ite

CFG with terminals in the theory
(+ input variables)

Example: Conditional LIA expressions w/o sums

E ::= x | ite C E E
C ::= E ≤ E | C ∧ C | ¬C
The SyGuS Project

SyGuS problem = ⟨theory, spec, grammar⟩

A first-order logic formula over the theory

By Example:
\[
\begin{align*}
  f(0, 1) &= 1 \\
  f(1, 0) &= 1 \\
  f(1, 1) &= 1 \\
  f(2, 0) &= 2
\end{align*}
\]

With free variables:
\[
\begin{align*}
  x &\leq f(x, y) \\
  y &\leq f(x, y) \\
  (f(x, y) = x \lor f(x, y) = y)
\end{align*}
\]
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?

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

- **depth ≤ 1**
  - \( N(1) = 1 \)

- **depth ≤ 2**
  - \( N(2) = 2 \)

- **depth ≤ 3**
  - \( N(3) = 5 \)

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

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

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

\[ N(d) \sim c^{2^d} \]

\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?

\[ E :::= x_1 \mid \ldots \mid x_k \mid E \oplus_1 E \mid \ldots \mid E \oplus_m E \]

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

- \( N(1) = 3 \)
- \( N(2) = 30 \)
- \( N(3) = 2703 \)
- \( N(4) = 21918630 \)
- \( N(5) = 1441279023230703 \)
- \( N(6) = 6231855668414547953818683022630 \)
- \( N(7) = 116508075215851596766492219468227024724121520304443212304350703 \)
- \( N(8) = \ldots \)

(k = m = 3)

Answer:
Pretty darn big.
Top-down Enumeration

- Start from initial symbol
- Repeatedly expand nonterminals using productions

\[
\text{L ::= sort(L) | L[N..N] | L + L | [N] | x} \\
\text{N ::= find(L,N) | 0} \\
\{[1,4,0,6] \rightarrow [1,4]\}
\]

\[
\text{top-down (⟨T, V, R, S⟩, \{i \rightarrow o\})} \\
\text{P := [S]} \\
\text{while (P \neq [])} \\
\text{p = P.dequeue();} \\
\text{if (ground(p) \land p([i]) = [o])} \\
\text{return p;} \\
\text{P.enqueue(expand(p))}
\]

\[
\text{expand(p, R)} \\
\text{L \rightarrow [sort(L), L[N..N], L + L, x]} \\
\text{L[N..N] \rightarrow [sort(L)[N..N], L[N..N][N..N], ... , L[0..N], L[N..0]]}
\]
Bottom-up Enumeration

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

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

{[1,4,0,6] → [1,4]}
```

```
bottom-up (⟨T, V, R, S⟩, {i → o})

P := [t | t in T ∧ t is nullary]

while (true)
    P += grow(P);

forall (p in P)
    if (S ~ p ∧ p([i]) = [o])
        return p;
```

```
grow (P, R)
N ~ 0 ∈ P L ::= [N] ∈ R
L ~ [0]
N ~ 0, L ~ [0] ∈ P N ::= find(L,N) ∈ R
N ~ find([0], 0)
```
How to make it scale

**Pruning**
Discard useless subprograms

\[ P = \{ \text{more promising candidates first} \} \]

\[ m \cdot N^2 \quad m \cdot (N-1)^2 \]

**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

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

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

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

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

...
Equivalent Subprograms

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

downward-upward
Equivalent Subprograms

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

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

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

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

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

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

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

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

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

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

\[\ldots\]
Observational Equivalence

\{[0] \rightarrow [0]\}\ \\
\text{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.

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

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

• Input:

\[
\begin{align*}
    \text{sort(sort(l))} &= \text{sort(l)} \\
    (l + l) + l &= l + (l + l) \\
    n &= n + 0 \\
    n + m &= m + n
\end{align*}
\]

Term Rewriting System (TRS)

\[
\begin{align*}
    \text{sort(sort(l))} &\rightarrow \text{sort(l)} \\
    (l + l) + l &\rightarrow l + (l + l) \\
    n + 0 &\rightarrow n \\
    n + m &\rightarrow^{(n > m)} m + n
\end{align*}
\]

Equivalences

\[
\begin{align*}
    \text{sort(x)} &= x[0..0] \times + x[0] \text{ find(x,0)} \\
    \text{sort(sort(x))} &\text{ rule 1 applies, not normal}
\end{align*}
\]
Lab #9

• Regular Expressions

  \[ R ::= \varepsilon \mid . \mid a \mid R \cdot R \mid R+R \mid R^* \]

- a → 1
- ab → 1
- ba → 0

assert match("a") == 1
assert match("ab") == 1
assert match("ba") == 0

Harness

generator bit[W] gen(bit[W] x, int depth) {
    assert depth > 0;
    if (??) return x;
    if (??) return ??;
    if (??) return ~gen(x, depth-1);
    if (??) {
        return { | gen(x, depth-1)
                  (+ | & | ^)
                  gen(x, depth-1) |};
    }
}

Sketch Regens