Syntax
Guided
Synthesis

Programming by Example

Synthesis

Guided
Reasoning

λ-calculus

Dependent
Types

Type
Theory
(basics)

Axiomatic
Semantics

Satisfiability
Modulo
Theory

Type
Directed
Synthesis

Refinement
Types

Counterexample
Guided
Inductive
Synthesis

Type
Directed
Synthesis

Synthesis!

MIT & NASA, 1957

“Code”

≈165cm

Syntax-Guided Synthesis (= SyGuS)

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

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

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

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

CFG FTW

- Context-free grammar

starting nonterminal

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

nonterminals

[4] terminals

productions

[5]
CFGs as Structural Constraints

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

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

The SyGuS Project

SyGuS problem = \langle theory, spec, grammar \rangle

\begin{itemize}
    \item A "library" of types and function symbols
    \item CFG with terminals in the theory (+ input variables)
    \item Example: Conditional LIA expressions w/o sums
\end{itemize}

Example: Linear Integer Arithmetic (LIA)

\begin{itemize}
    \item True, False
    \item 0, 1, 2, ...
    \item \( x, y, z, +, *, \text{ite} \)
\end{itemize}

The SyGuS Project

SyGuS problem = \langle theory, spec, grammar \rangle

\begin{itemize}
    \item A first-order logic formula over the theory
    \item By Example:
        \[ f(0, 1) = 1 \wedge f(1, 0) = 1 \wedge f(1, 1) = 1 \wedge f(2, 0) = 2 \]
    \item With free variables:
        \[ x \leq f(x, y) \wedge y \leq f(x, y) \wedge f(x, y) = x \vee f(x, y) = y \]
\end{itemize}

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

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?

$$E ::= \{x \mid \ldots \mid x \} \cup E \cup E$$

$$N(d) = 1 + N(d - 1)^2$$

- depth ≤ 1
  - $$N(1) = 1$$
- depth ≤ 2
  - $$N(2) = 2$$
- depth ≤ 3
  - $$N(3) = 5$$

$$N(d) = 1 + N(d - 1)^2$$

$$N(1) = 1$$
$$N(2) = 2$$
$$N(3) = 5$$
$$N(4) = 26$$
$$N(5) = 677$$
$$N(6) = 45833$$
$$N(7) = 210066388901$$
$$N(8) = 44127887745906175987802$$
$$N(9) = 1947270476915296449559703445493848930452791205$$
$$N(10) = 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026$$

**Answer:**
Pretty darn big.

---

**Top-down Enumeration**

- Start from initial symbol
- Repeatedly expand nonterminals using productions

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

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

$$L \implies \{1,4,6\} \to \{1,4\}$$

**Bottom-up Enumeration**

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

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

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

$$L \implies \{1,4,6\} \to \{1,4\}$$
How to make it scale

Pruning
Discard useless subprograms

Ranking
Explore more promising candidates first

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

\[ m \cdot N^2 \]

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

• Explore more promising candidates first
• Discard useless subprograms

\[ N \cdot N \]

\[ N \cdot (N-1) \]

\[ m \cdot N^2 \]

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

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

\[ \text{sort}(x) \]

\[ \text{sort}(\text{sort}(x)) \]

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

\[ \text{x} + \text{x} \]

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

\[ \text{sort}(\text{sort}(x)) \]

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

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

\[ \text{sort}([0]) \]

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

\[ \text{x} + \text{x}[0..0] \]

\[ \text{x} + (\text{x} + \text{x}) \]

\[ \text{x} + [0] \]

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

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

\[ \text{x}[0..0] + \text{x} \]

\[ (\text{x} + \text{x}) + \text{x} \]

\[ [0] + \text{x} \]

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

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

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

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

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

\[ [0][0..0] \]

...
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...

\[ p_1 \equiv p_2 \quad \Rightarrow \quad p_1[[i]] = p_2[[i]] \]

Observational Equivalence

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

Observational Equivalence

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

Observational Equivalence

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

• ESolver [Udupa et al. ’13]
• Escher [Alur et al. ‘17]
• Lens [Phothilimthana et al. ’16]
• EuSolver [Alur et al. ‘17]

Used in almost all PBE tools:
Built-in Equivalences

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

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

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

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

26

User-specified Equivalences

• Input:

\[
\text{sort(sort(l))} \rightarrow \text{sort(l)} \\
(l + l) + l \rightarrow l + (l + l) \\
n + 0 \rightarrow n \\
(n > m) + n \rightarrow (n > m) + n
\]

Term Rewriting System (TRS)

27

Lab #8

• We want to synthesize regular expressions

\[
a \rightarrow 1 \\
ab \rightarrow 1 \\
ba \rightarrow 0
\]

Proxy representation: Finite State Machine

28

Harness

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