Syntax-Guided Synthesis
Enumerative Search

CS560: Reasoning About Programs

Roopsha Samanta

Based on slides by Rajeev Alur, Nadia Polikarpova, Armando Solar-Lezama, Xiaokang Qiu
Roadmap

Previously
- Logics for reasoning about programs
- Verification and analysis of programs

Today
- Syntax-guided synthesis
- Inductive program synthesis
- Enumerative search
Post 2000: Modern Program Synthesis
Transformational program synthesis: A *search* problem

**Specification**
- Input-output examples
- Logical specifications
- Equivalent programs
- Natural language

*Domain-specific* search space

**Programs**
- Find a program in search space consistent with specification
Dimensions in modern program synthesis

- **User intent**: How do you tell the system what you want?
- **Search strategy**: How does the system find the program you actually want?
- **Search space**: What is the space of programs to explore?
Dimensions in modern program synthesis

User intent
- Input-output examples
- Logical specifications
- Equivalent programs
- Natural language

Search strategy
- Enumerative search
- Representation-based search
- Constraint-based search
- Stochastic search
- ML-based

Search space
- Grammars
- DSLs
- Generators
- Components
Dimensions in modern program synthesis

User intent
- Input-output examples
  - Logical specifications
  - Equivalent programs
  - Natural language

Search strategy
- Enumerative search
- Representation-based search
- Constraint-based search
- Stochastic search
- ML-based

Search space
- Grammars
- DSLs
- Generators
- Components
Example

Input-output example:

1,4,2,0,7,9,2,5,0,3,2,4,7 → 1,2,4,0

Context-free grammar:

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

\[
N ::= \text{find}(L,N) \mid \text{len}(L) \mid 0 \mid N + 1
\]

Synthesized program:

\[
f(x) := \text{sort}(x[0, \text{find}(x,0)]) + [0]
\]
Dimensions in modern program synthesis

- User intent
- Input-output examples
- Search strategy
- Enumerative search

- Search space
- Grammars
Context-free Grammars (CFGs): <T, N, R, S>

S: Starting non-terminal

N: Non-terminals

T: Terminals

R: Rules/productions

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

N ::= find(L,N) | len(L) | 0 | N + 1
Context-free Grammars (CFGs): \( <T, N, R, S> \)

\[
L ::= \text{sort}(L) \quad | \quad L[N,N] \quad | \quad L + L \quad | \quad [N] \quad | \quad x
\]

\[
N ::= \text{find}(L,N) \quad | \quad \text{len}(L) \quad | \quad 0 \quad | \quad N + 1
\]
SYntax-GUIDed Synthesis (SyGuS)

Core computational problem: Find a program $P$ such that
1. $P$ is in a set $E$ of programs (syntactic constraint)
2. $P$ satisfies spec $\varphi$ (semantic constraint)

Common theme to many recent efforts
- Sketch (Bodik, Solar-Lezama et al)
- FlashFill (Gulwani 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)
- Implicit programming: Scala$^*$Z3 (Kuncak et al)
- Auto-grader (Singh et al)

But no way to share benchmarks and/or compare solutions!
SyGuS Setup

Fix a background theory $T$: fixes types and operations

Function/expression to be synthesized: name $f$ along with its type

Inputs to SyGuS problem:
- Specification $\varphi$:
  *Typed formula using symbols in $T$ + symbol $f$*
- Set $E$ of expressions given by a CFG:
  *Set of candidate expressions that use symbols in $T$*

Computational problem:
Output $e$ in $E$ such that $\varphi[f/e]$ is valid modulo theory $T$
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.
Example

- Theory QF-LIA

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

- Specification: $(x \leq f(x,y)) \ & \ (y \leq f(x,y)) \ & \ (f(x,y) = x \ | \ f(x,y) = y)$

- Candidate Implementations: Conditional expressions without +

  Term := x | y | Const | If-Then-Else (Cond, Term, Term)
  Cond := Term <= Term | Cond & Cond | ~ Cond | (Cond)

- Possible solution:
  If-Then-Else $(x \leq y, \ y, x)$

What about example-based specs?
Example

- Theory QF-LIA

- Function to be synthesized \( f \) (int \( x \), int \( y \)) : int

- Specification: \( f(0,1) = 1 \) & \( f(1,0) = 1 \)

- Candidate Implementations: Conditional expressions without +

  \[
  \text{Term} := x | y | \text{Const} | \text{If-Then-Else} (\text{Cond}, \text{Term}, \text{Term})
  \]

  \[
  \text{Cond} := \text{Term} \leq \text{Term} | \text{Cond} \& \text{Cond} | \sim \text{Cond} | (\text{Cond})
  \]

- Possible solution:

  \( \text{If-Then-Else} (x \leq y, y, x) \)
- Annual SyGuS competition
- Standardized input language (SYNTH-LIB)
- Benchmarks

https://sygus.org/
Dimensions in modern program synthesis

User intent
Input-output examples
Search strategy
Enumerative search

Search space
Grammars
Inductive synthesis
Programming by example
Example-based synthesis
Inductive programming

Synthesize a program whose behavior satisfies a set of examples
∀x,y,z.
\[ x \leq \max(x,y,z) \land y \leq \max(x,y,z) \land z \leq \max(x,y,z) \land (\max(x,y,z)=x \lor \max(x,y,z)=y \lor \max(x,y,z)=z) \]

```
int max (int x,int y,int z)
int m = z;
if (z <= y) m = y;
if (m <  x) m = x;
return m;
```
```c
int max (int x, int y, int z)
{
    int m = z;
    if (z <= y) m = y;
    if (m < x) m = x;
    return m;
}
```

Program/Search Space

- $(0, 10, 2) \mapsto 10$
- $(-1, 10, 20) \mapsto 20$
- $(-1, -2, -3) \mapsto -1$
Problems in inductive program synthesis

Program/Search Space

```
int max (int x, int y, int z)
int m = z;
if (z <= y) m = y;
if (m < x) m = x;
return m;
```

- (0, 10, 2) ↦ 10
- (-1, 10, 20) ↦ 20
- (-1, -2, -3) ↦ -1
Problems in inductive program synthesis

Program/Search Space

```
int max (int x, int y, int z) {
    int m = z;
    if (z <= y) m = y;
    if (m < x) m = x;
    return m;
}
```

```
int max (int x, int y, int z) {
    int m = x;
    if (y < z) m = z;
    if (m < y) m = y;
    return m;
}
```
Problems in inductive program synthesis

Program/Search Space

int max (int x, int y, int z)
    int m = z;
    if (z <= y) m = y;
    if (m < x) m = x;
    return m;

int max (int x, int y, int z)
    int m = x;
    if (y < z) m = z;
    if (m < y) m = y;
    return m;

Overfitting!
Problems in inductive program synthesis

(0, 10, 2) ⇔ 10
(-1, 10, 20) ⇔ 20
(-1, -2, -3) ⇔ -1

int max (int x, int y, int z)
int m = x;
if (y < z) m = z;
if (m < y) m = y;
return m;

Overfitting!
Problems in inductive program synthesis

Brittleness!

$$\begin{align*}
(0, 10, 2) & \mapsto 10 \\
(-1, 10, 20) & \mapsto 20 \\
(-1, -2, -3) & \mapsto -1
\end{align*}$$

$$\begin{align*}
\text{int } \text{max} \ (\text{int } x, \text{int } y, \text{int } z) \\
\text{int } m = x; \\
\text{if } (y < z) m = z; \\
\text{if } (m < y) m = y; \\
\text{return } m;
\end{align*}$$

$$\begin{align*}
(0, 10, 2) & \mapsto 10 \\
(-1, 10, 20) & \mapsto 20 \\
(-1, -3, -2) & \mapsto -1
\end{align*}$$

$$\begin{align*}
\text{int } \text{max} \ (\text{int } x, \text{int } y, \text{int } z) \\
\text{int } m = x; \\
\text{if } (y < z) m = z; \\
\text{if } (m < y) m = y; \\
\text{return } m;
\end{align*}$$
Problems in inductive program synthesis

(0, 10, 2) ↦ 10
(-1, 10, 20) ↦ 20
(-1, -2, -3) ↦ -1

How can these problems be addressed?
Problems in inductive program synthesis

(0, 10, 2) ↦ 10
(-1, 10, 20) ↦ 20
(-1, -2, -3) ↦ -1

How can these problems be addressed?

Careful design of search space + search strategy
Dimensions in modern program synthesis

User intent

Input-output examples

Search strategy

Enumerative search

Search space

Grammars
Enumerative search
Enumerative/explicit/exhaustive search

Key idea:
Generate programs from the grammar one by one and test on examples

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
Bottom-up enumeration

\[
\text{plist := set of all terminals}
\]

\[
\text{while true}
\]

\[
\text{plist := grow(plist);}
\]

\[
\text{forall p in plist}
\]

\[
\text{if isCorrect(p)}
\]

\[
\text{return p;}
\]

\[
\text{grow(plist)}
\]

\[
// return a list of all trees generated by
// taking a non-terminal and adding
// nodes in plist as children
\]
Bottom-up enumeration

L ::= sort(L)  |  L[N,N]  |  L + L  |  N
N ::= find(L,N)  |  len(L)  |  0  |  N + 1

Set grows very fast!
Large equivalence classes of equivalent programs
Pruning equivalent programs

- Program equivalence is hard
  - It is also unnecessary!

- Observational Equivalence
  - Are they equivalent w.r.t the inputs
    - easy to check efficiently
    - sufficient for the purpose of PBE
  - Keep only the simplest one

```plaintext
plist := set of all terminals
while true
    plist := grow(plist);
    plist := reduce(plist);
    forall p in plist
        if isCorrect(p)
            return p;
```
Enumerative 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 find, just need to be able to execute them
  - no need to know of any properties about them
    e.g. automatically ignores sort(sort(x)) without having to know that sort is idempotent
- Complexity depends on the size of the set of distinct programs
Enumerative 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:

- Recursive Program Synthesis [AGK13]
- EUSolver [Alur et al. 2017]
Summary

Today
- Inductive synthesis
- SyGuS
- Enumerative search

Next
- Representation-based search (Version Space Algebras)