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

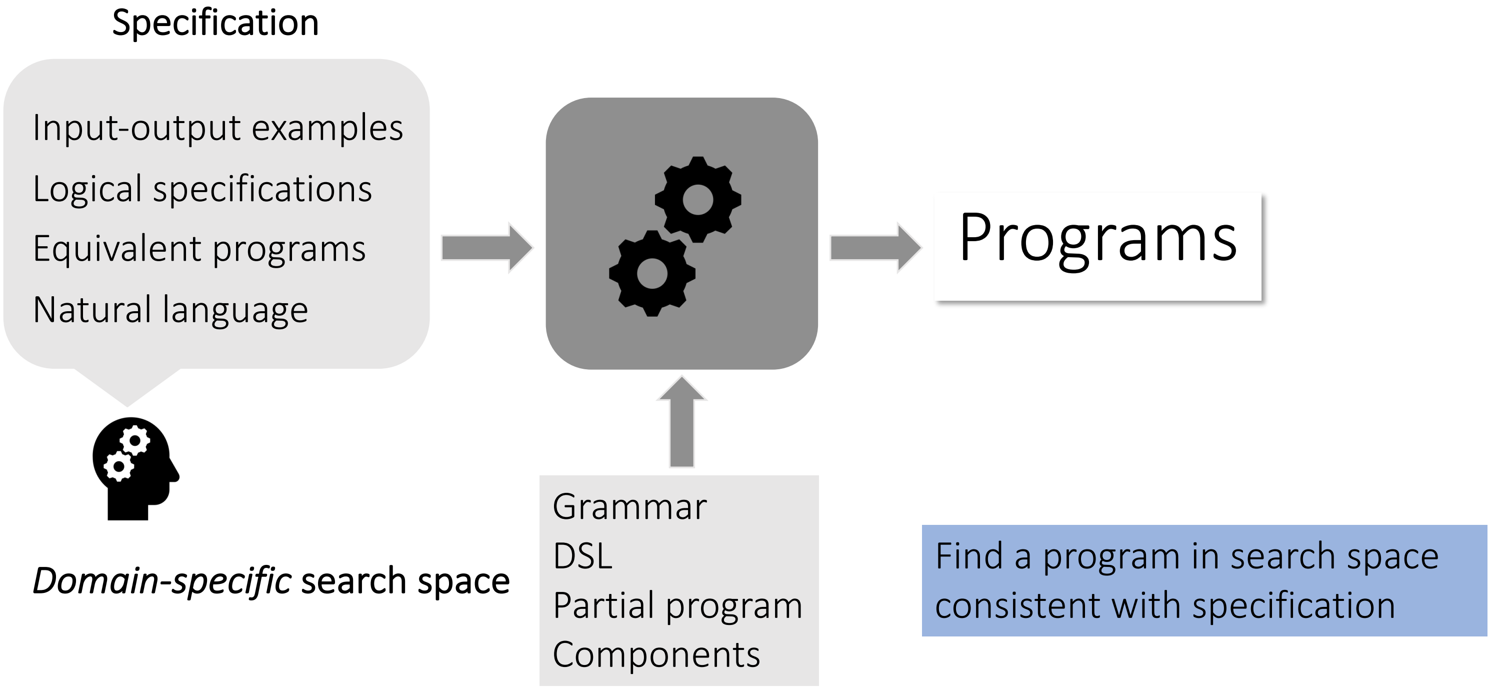
- Logics for reasoning about programs
- Verification and analysis of programs

Today

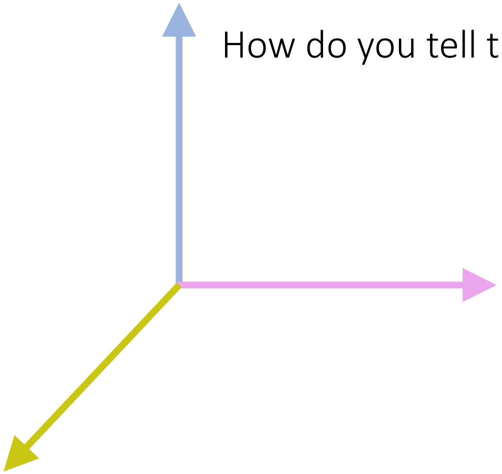
- Syntax-guided synthesis
- Inductive program synthesis
- Enumerative search

Post 2000: Modern Program Synthesis

Transformational program synthesis: A *search* problem



Dimensions in modern program synthesis



User intent

How do you tell the system what you want?

Search strategy

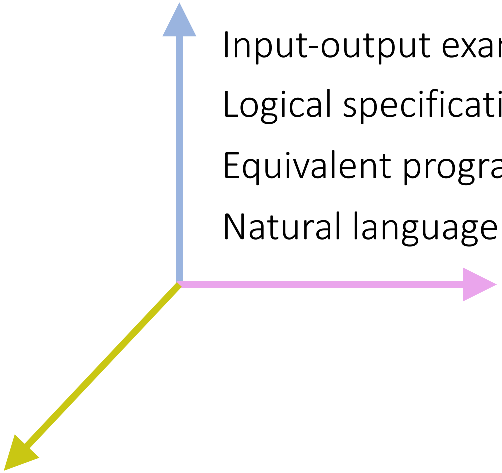
How does the system find the program you actually want?

Search space

What is the space of programs to explore?

[Gulwani 2010]

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

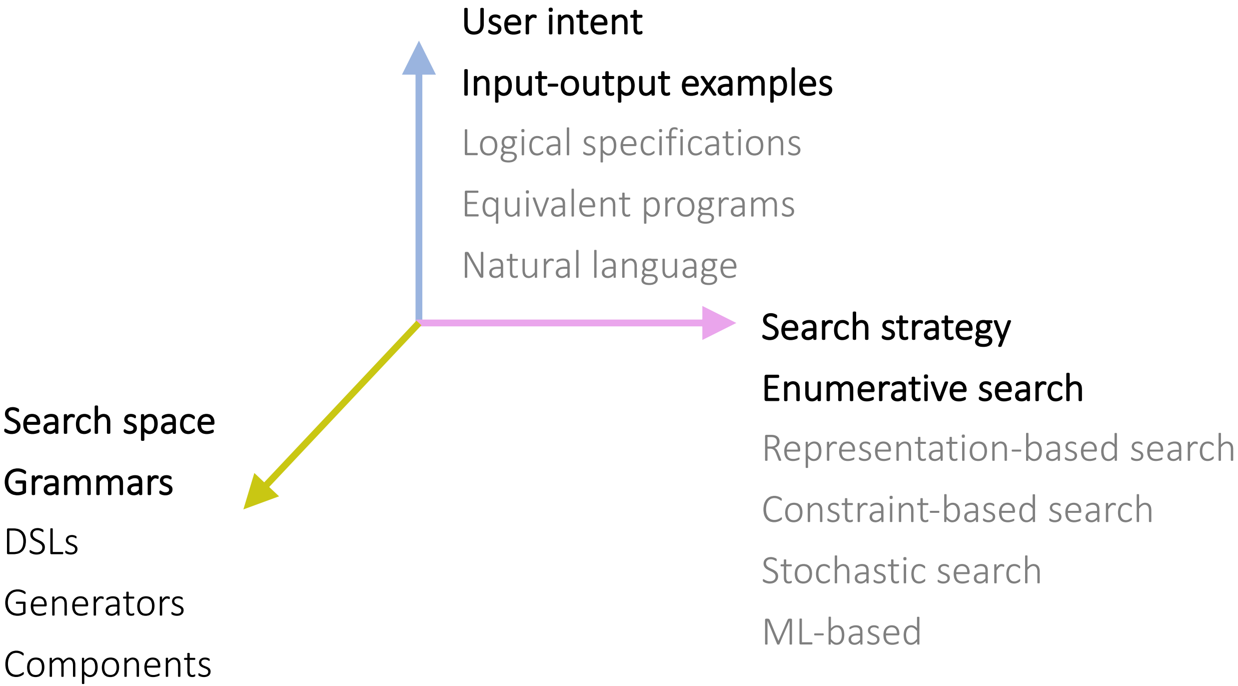
Grammars

DSLs

Generators

Components

Dimensions in modern program synthesis



Example

Input-output example:

```
1,4,2,0,7,9,2,5,0,3,2,4,7 → 1,2,4,0
```

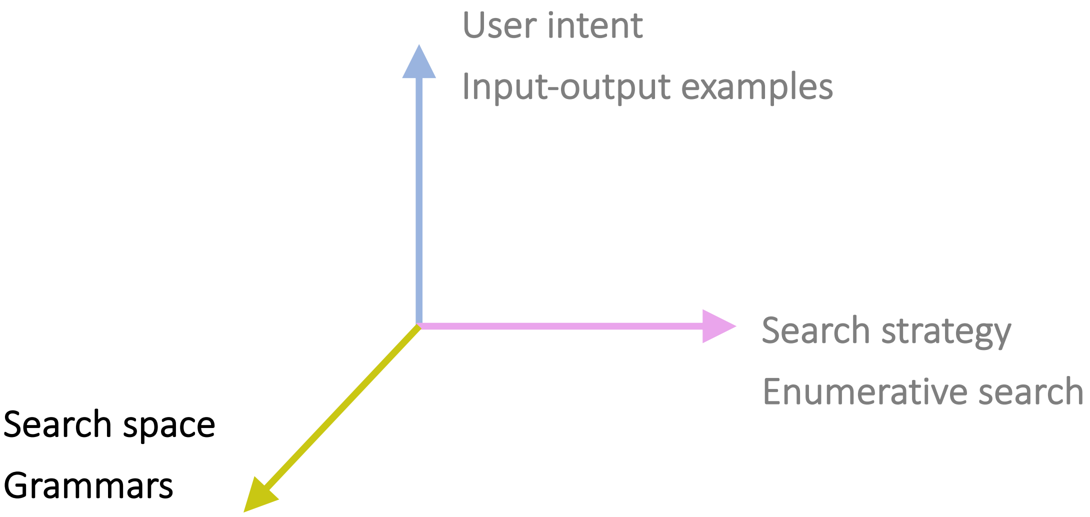
Context-free grammar:

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

Synthesized program:

```
f(x) := sort(x[0, find(x,0)]) + [0]
```

Dimensions in modern program synthesis



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

S: Starting non-terminal

N: Non-terminals

T: Terminals

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

R: Rules/productions

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

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

x

0, 1, 2, 3, . . .

x+x

sort(x[0, len(x)])

sort(x) + sort(x[1,3])

find(x, 0)

[3]

find(sort([1] + [x]), 1)

sort(x[0, find(x,5)])

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 φ (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

No programs with loops

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

Inputs to SyGuS problem:

- Specification φ:

  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

Expression grammar

Computational problem:

Output e in E such that φ[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 (int x, int y) : int

- Specification: (x ≤ f(x,y)) & (y ≤ f(x,y)) & (f(x,y) =x | f(x,y)=y)

- Candidate Implementations: Linear expressions

  LinExp := x | y | Const | LinExp + LinExp | LinExp - LinExp

- No solution exists

Example

- Theory QF-LIA
- Function to be synthesized f (int x, int y) : int
- Specification: $(x \leq f(x,y))$ & $(y \leq f(x,y))$ & $(f(x,y) = x \mid f(x,y) = y)$
- 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 +

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

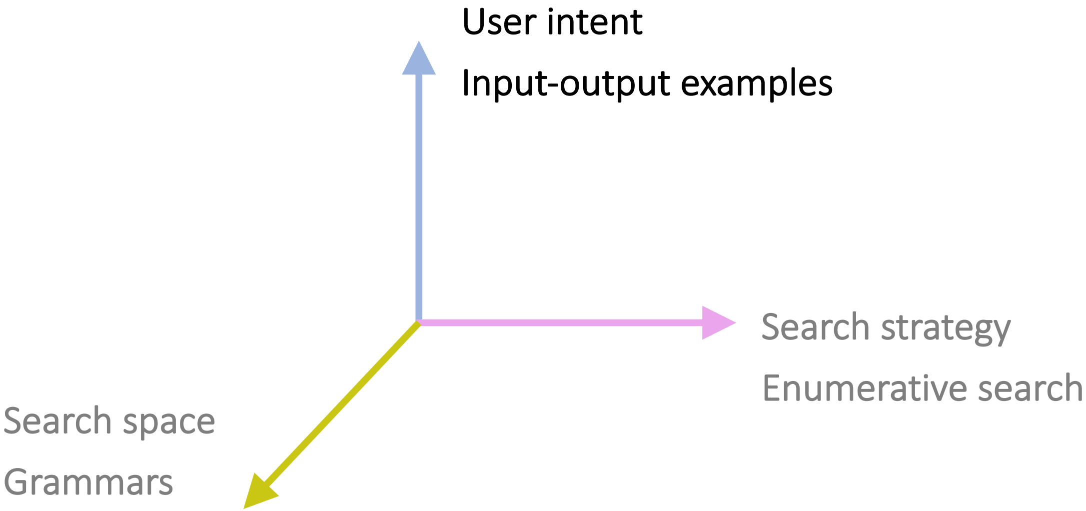
- Possible solution:
  If-Then-Else ($x \leq y$, y, x)

SYNT Workshop @ CAV

- Annual SyGuS competition
- Standardized input language (SYNTH-LIB)
- Benchmarks

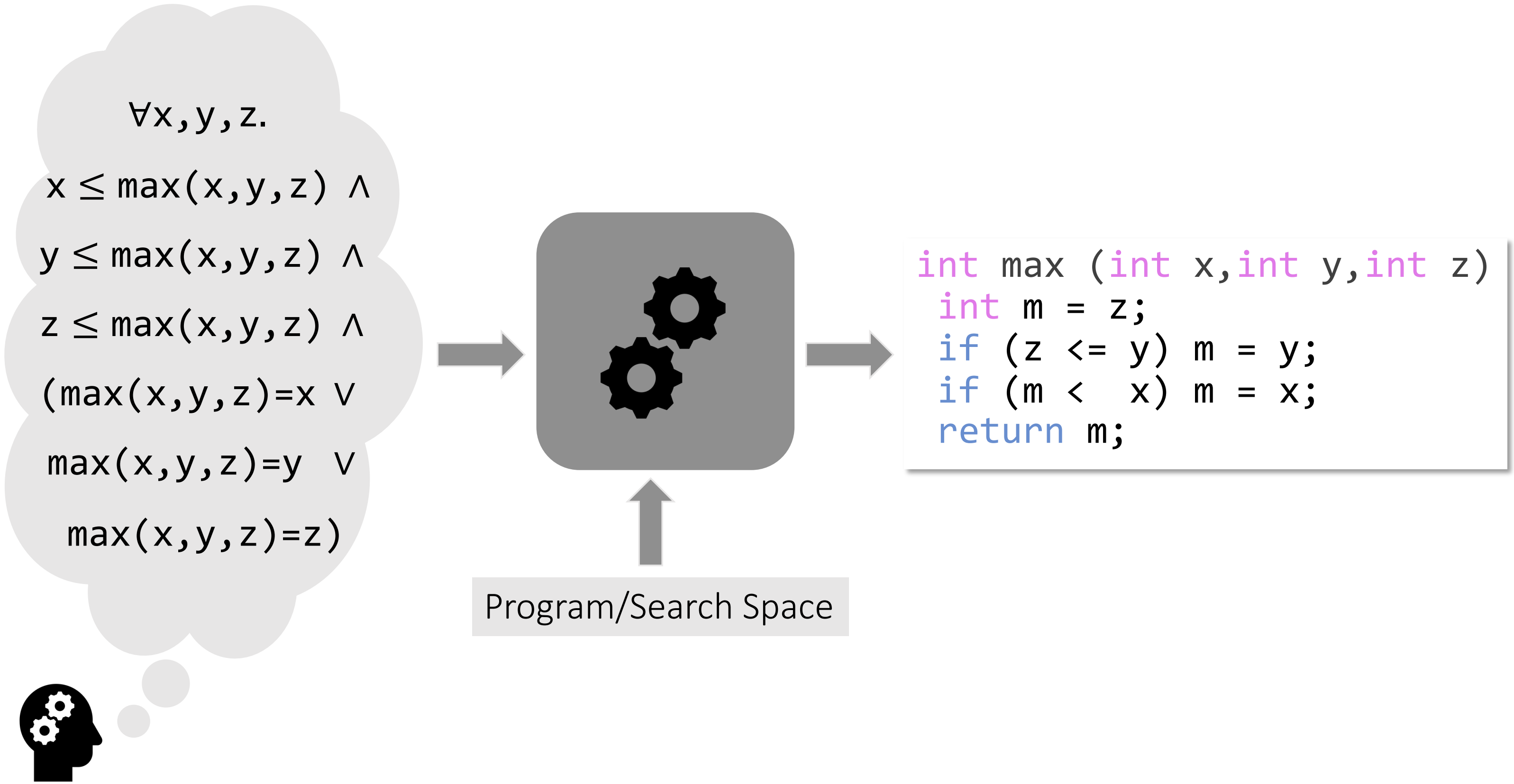
https://sygus.org/

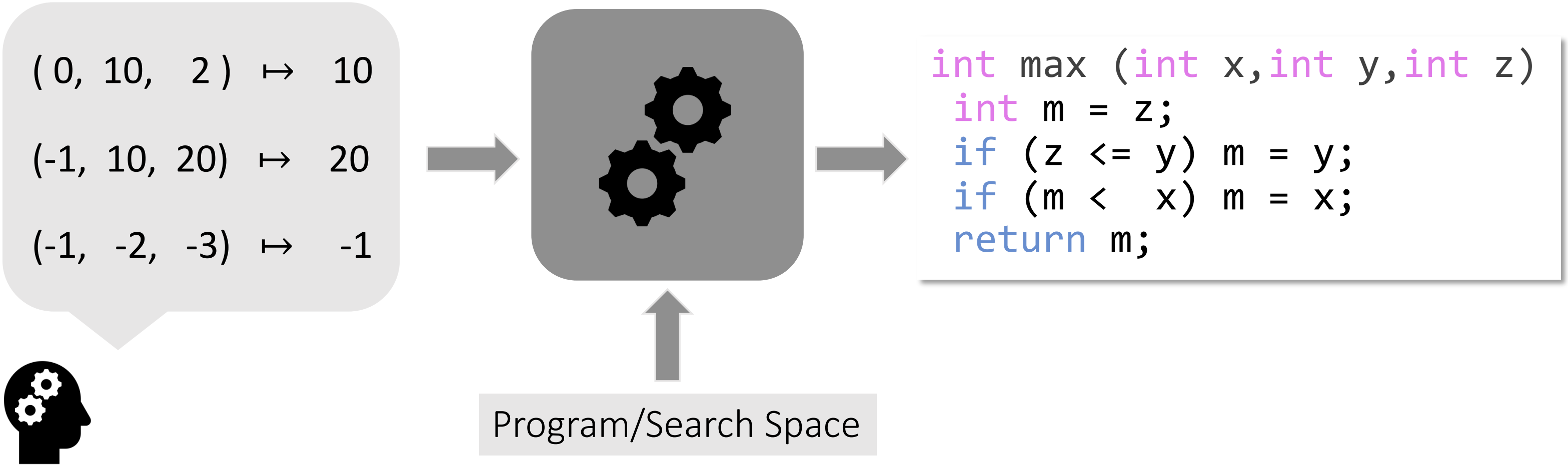
Dimensions in modern program synthesis



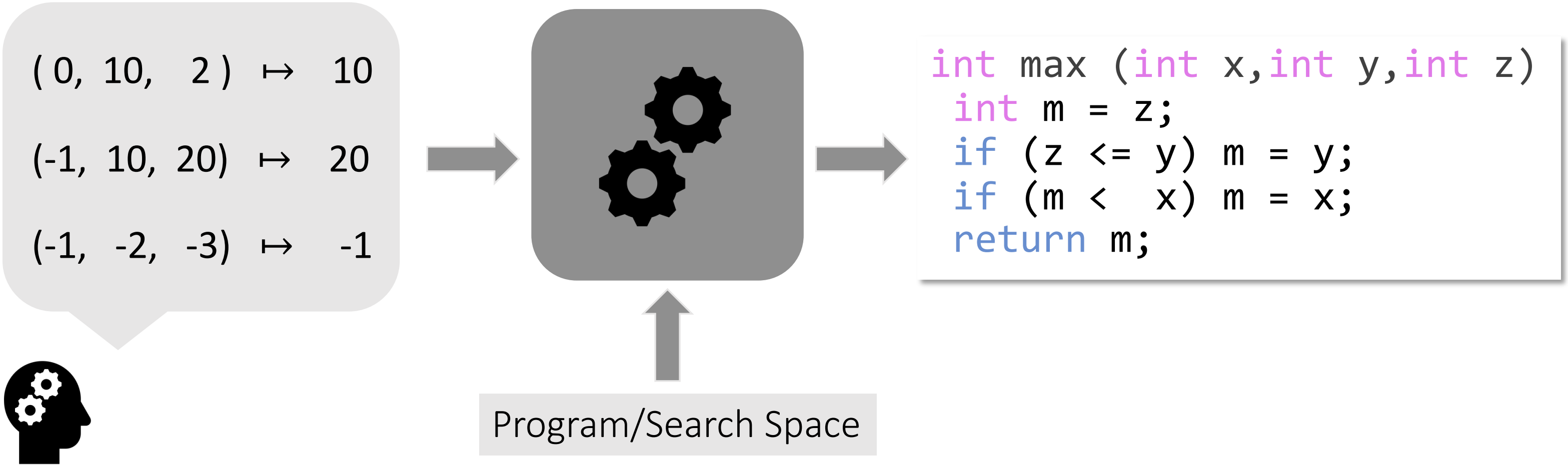
Inductive synthesis
Programming by example (s)
Example-based synthesis
Inductive programming

Synthesize a program whose behavior satisfies a set of examples


∀x,y,z.
x ≤ max(x,y,z) ∧
y ≤ max(x,y,z) ∧
z ≤ max(x,y,z) ∧
(max(x,y,z)=x ∨
max(x,y,z)=y ∨
max(x,y,z)=z)
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
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

Problems in inductive program synthesis



Problems in inductive program synthesis

(0, 10, 2)	$\mapsto$	10
(-1, 10, 20)	$\mapsto$	20
(-1, -2, -3)	$\mapsto$	-1

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;
```

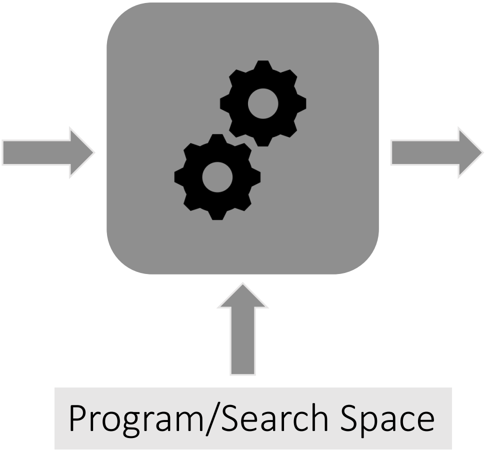
Ambiguity!

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 = 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;
```

Program/Search Space

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;
```

✗

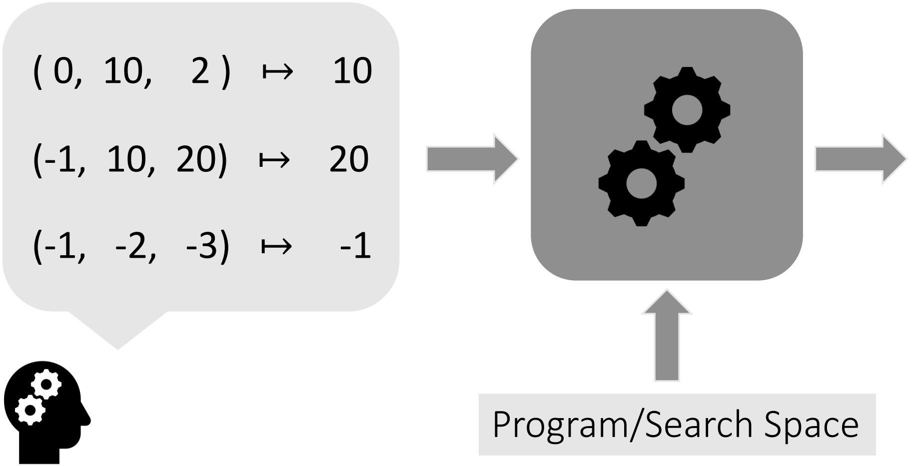
(0, 10, 2) ↦ 10
(-1, 10, 20) ↦ 20
(-1, -3, -2) ↦ -1

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

✓

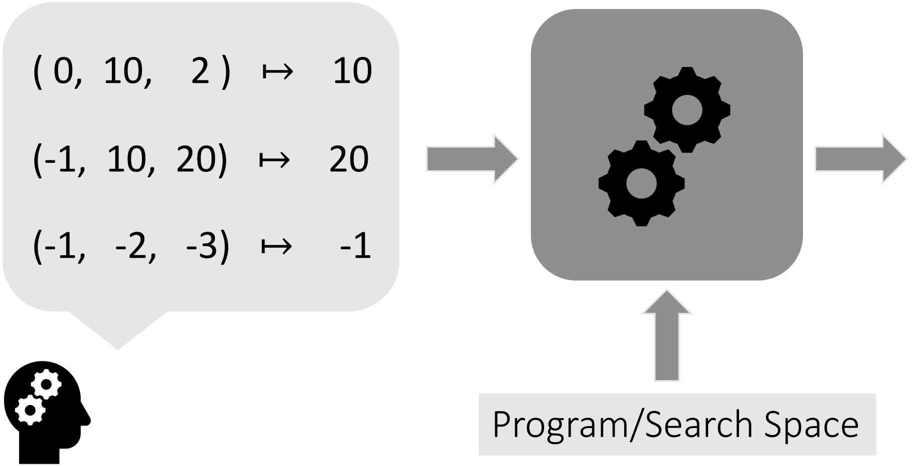
Brittleness!

Problems in inductive program synthesis



How can these problems be addressed?

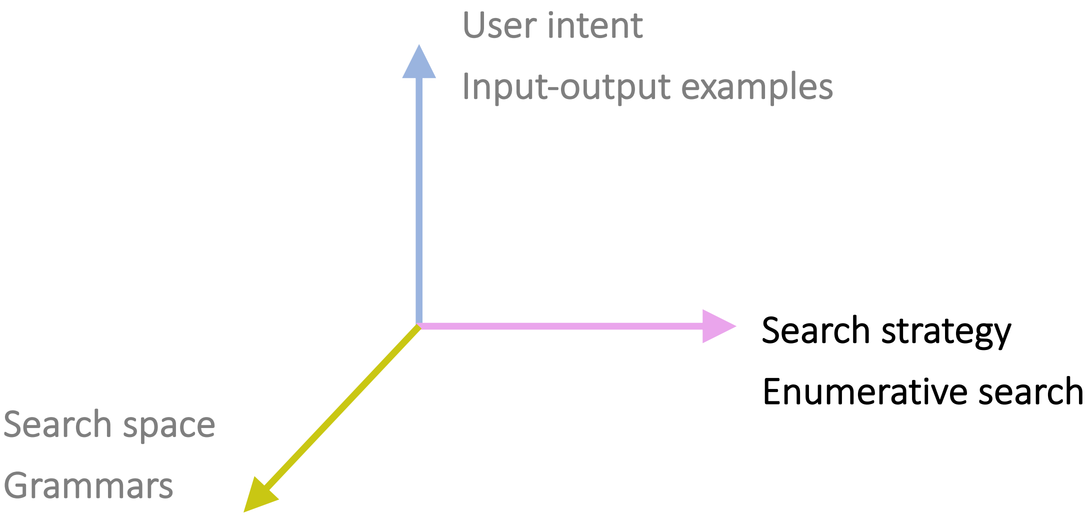
Problems in inductive program synthesis



How can these problems be addressed?

Careful design of search space + search strategy

Dimensions in modern program synthesis



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

Top-down enumeration

```
plist := set of all terminals
while true
    plist := grow(plist);
    forall  p in plist
        if isCorrect(p)
           return p;


grow(plist)
  // return a list of all trees generated by
  // taking a non-terminal and adding
  // nodes in plist as children
```

Starting from terminals, combine sub-programs into larger programs using productions

Bottom-up enumeration

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

Level 0   x   0

Level 1

Level 1

How many programs at level d?

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

```
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)