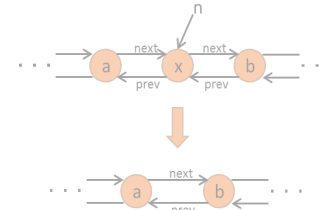
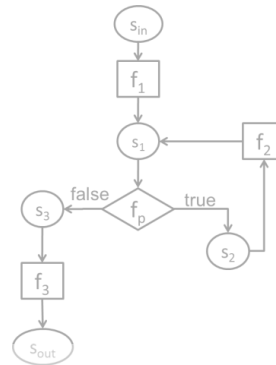


$$\exists c \forall in Q(c, in)$$

```

/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
    int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
    assert t == (x+y)/2;
    return t;
}

```

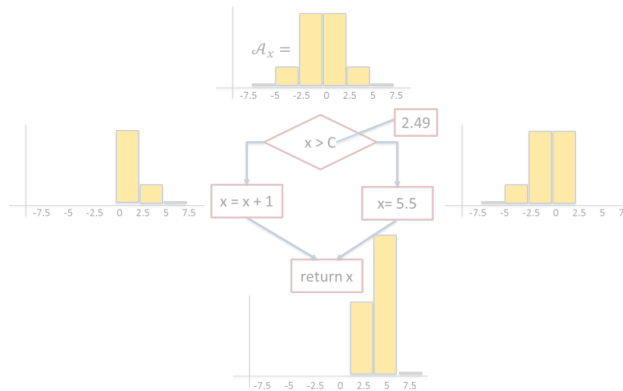
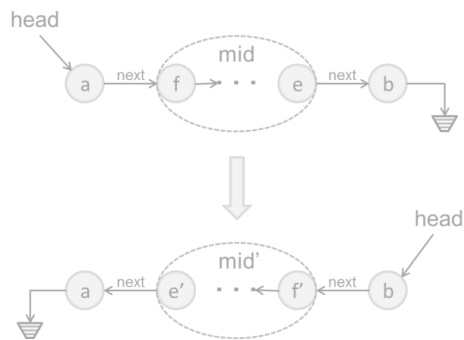


```

{
    s = n.succ;
    p = n.pred;
    p.succ = s;
    s.pred = p;
}

```

Module I: Searching for Simple Programs



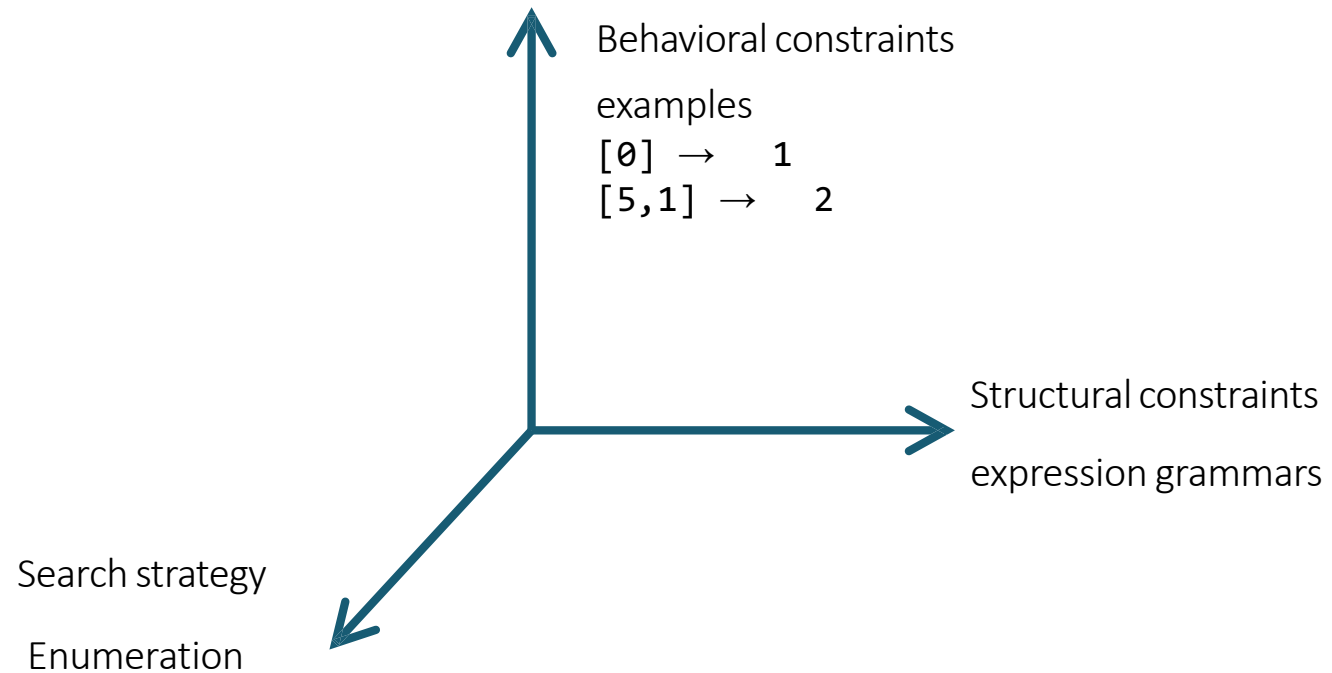
$$\varphi(p)$$

$$Sk[c](in)$$

Lecture 2

Syntax-Guided Synthesis and Enumerative Search

Week 1-2



Today

Synthesis from examples: motivation and history

Syntax-guided synthesis

- expression grammars as structural constraints
- the SyGuS project

Enumerative search

- enumerating all programs generated by a grammar
- bottom-up vs top-down

Synthesis from examples

Synthesis from Examples

=

Programming by Example

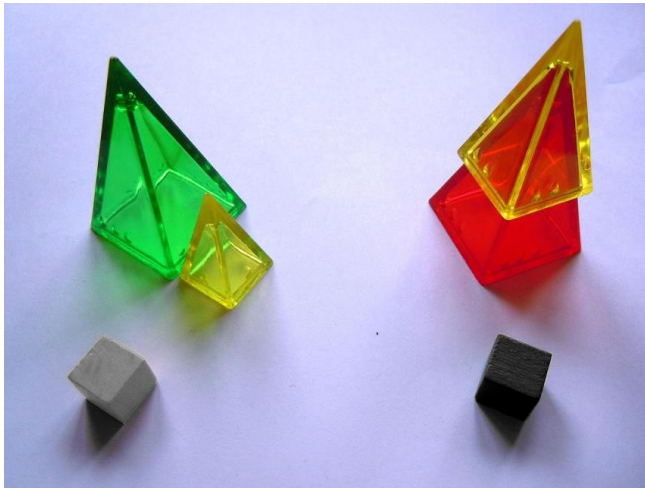
=

Inductive Synthesis

Inductive Programming

Inductive Learning

The Zendo game



This is called inductive learning!

The **teacher** makes up a secret rule

- e.g. all pieces must be grounded

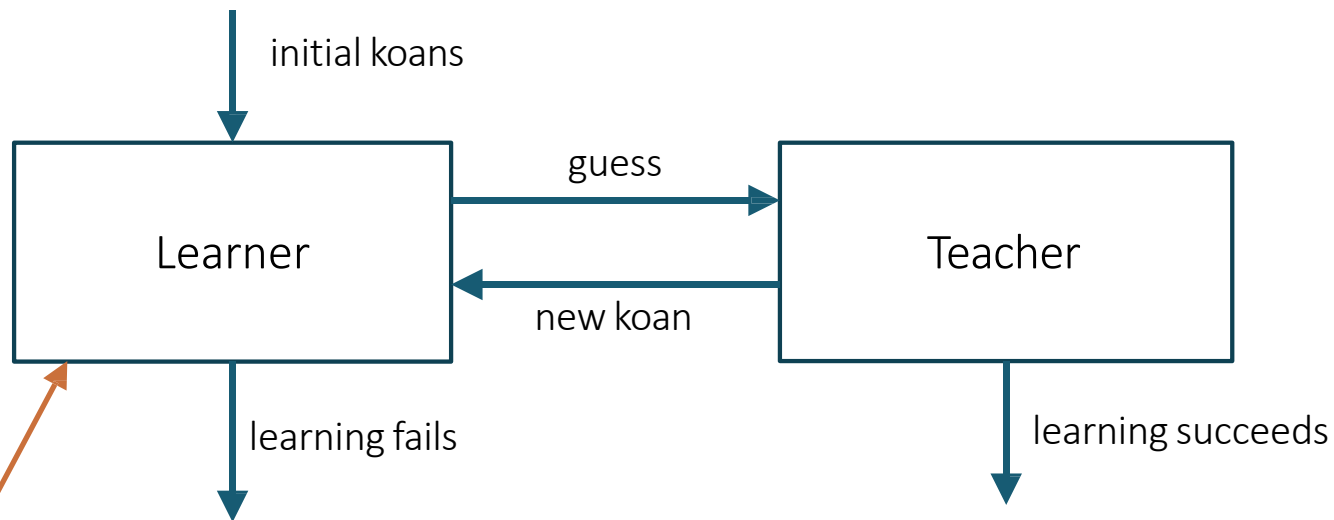
The teacher builds two **koans** (a positive and a negative)

Students take turns to build koans and ask the teacher to label them

A student can try to guess the rule

- if they are right, they win
- otherwise, the teacher builds a koan on which the two rules disagree

The Zendo game



1960s: humans are good at this...
can computers do this?

A little bit of history: inductive learning

MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970



Patrick
Winston

Explored the question of generalizing from a set of observations

- Similar to Zendo

Became the foundation of machine learning

A little bit of history: PBE/PBD

1980s: searching a predefined list of programs

1990s (Tessa Lau): bring inductive learning techniques into PBE

Programming by Demonstration: An Inductive Learning Formulation*

Tessa A. Lau and Daniel S. Weld
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
October 7, 1998
{tlau, weld}@cs.washington.edu

ABSTRACT

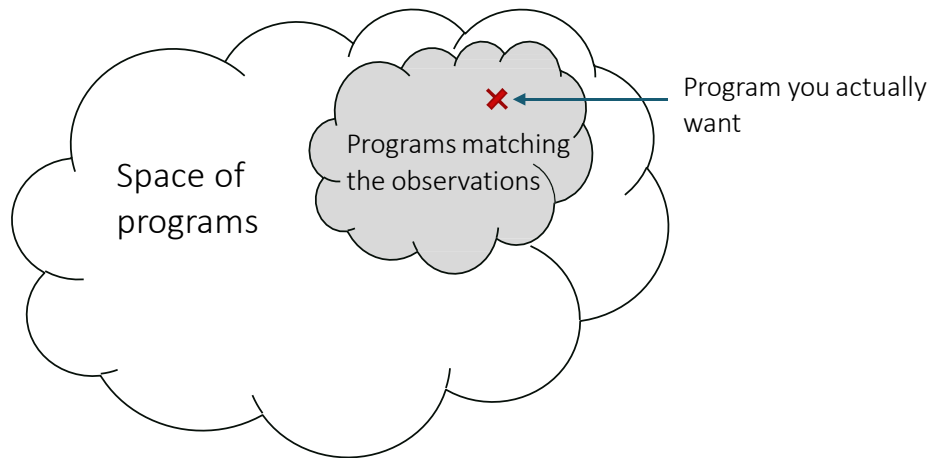
Although Programming by Demonstration (PBD) has

- Applications that support macros allow users to record a fixed sequence of actions and later replay this sequence using a shortcut such as a mouse click or a



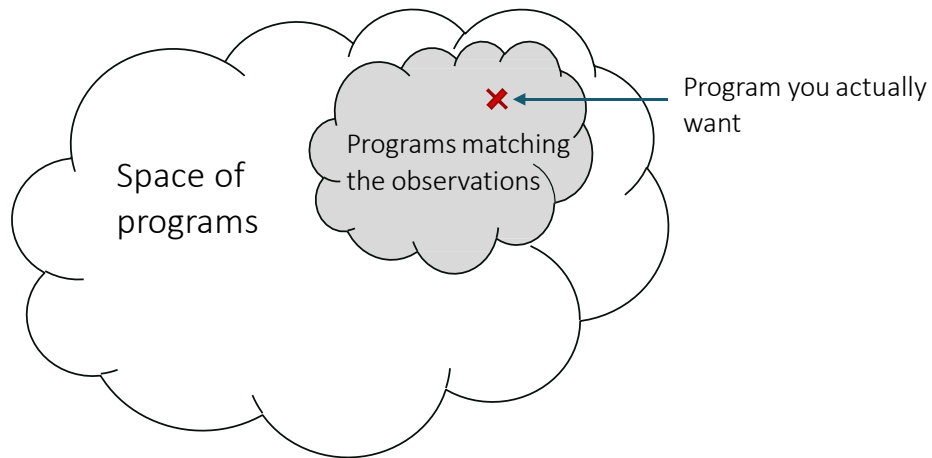
Tessa Lau

Key issues in inductive learning



- (1) How do you find a program that matches the observations?
- (2) How do you know it is the program you are looking for?

Key issues in inductive learning



Traditional ML emphasizes (2)

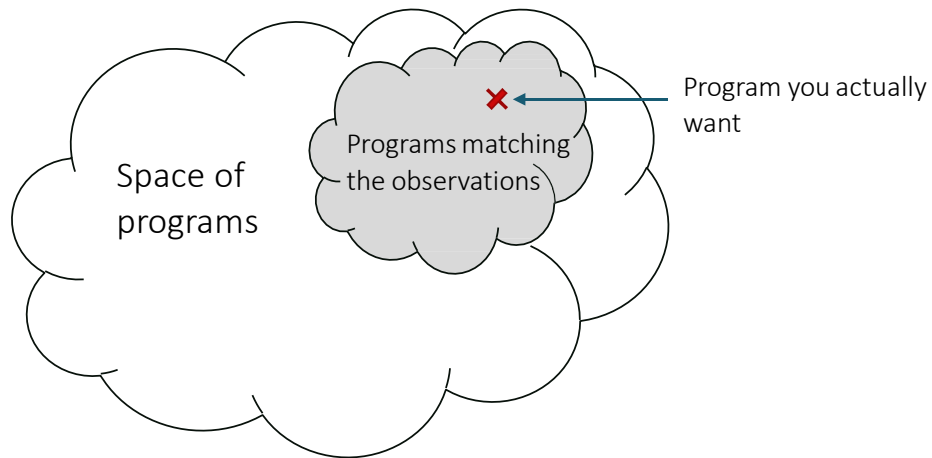
- Fix the space so that (1) is easy

So did a lot of PBD work

(1) How do you find a program that matches the observations?

(2) How do you know it is the program you are looking for?

The synthesis approach

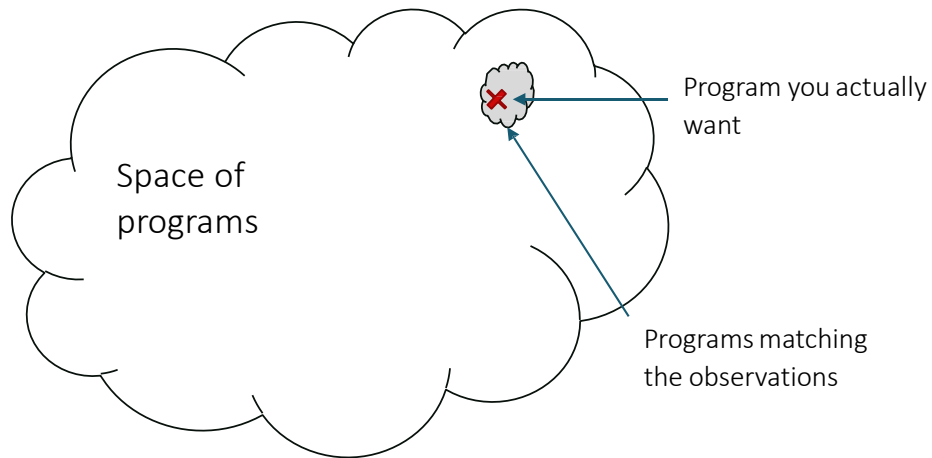


Modern emphasis

(1) How do you find a program that matches the observations?

(2) How do you know it is the program you are looking for?

The synthesis approach



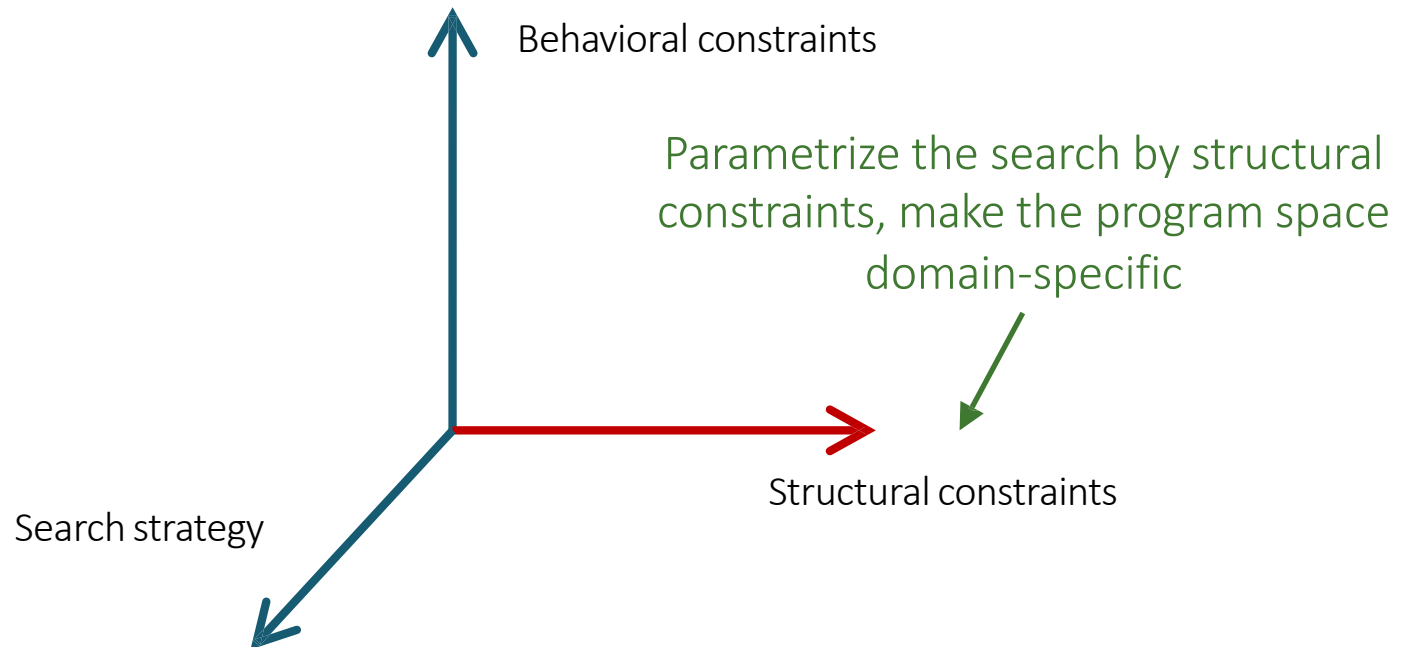
Modern emphasis

- If you can do really well with (1) you can win
- (2) is still important

(1) How do you find a program that matches the observations?

(2) How do you know it is the program you are looking for?

Key idea



Syntax-Guided Synthesis

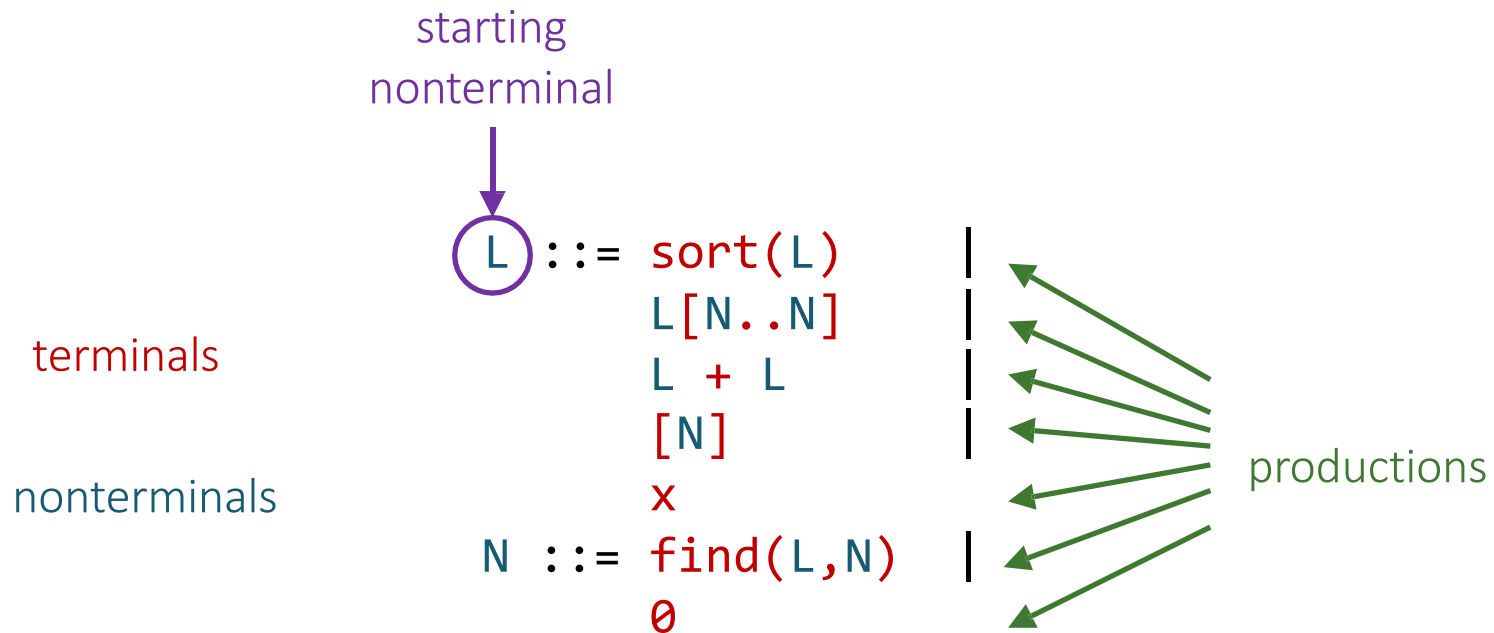
Example

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

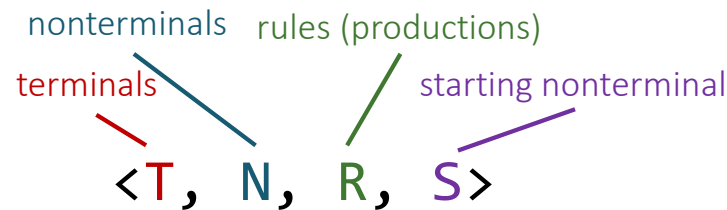
$f(x) := \text{sort}(x[0..\text{find}(x, 0)]) + [0]$

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

Context-free grammars (CFGs)



Context-free grammars (CFGs)



Sentential forms: $\{\alpha \in (N \cup T)^*\}$

Rewrites to: $\alpha A \beta \rightarrow \alpha \gamma \beta \equiv (A \rightarrow \gamma) \in R$

(Incomplete) terms/programs: $\{\alpha \in \{N \cup T\}^* \mid A \rightarrow^* \alpha\}$

Ground programs: $\{\alpha \in T^* \mid A \rightarrow^* \alpha\}$

= programs without holes, complete programs

Whole programs: $\{\alpha \in T^* \mid S \rightarrow^* \alpha\}$

= roughly, programs of the right type

x N x

x N x -> x find(L,N) x

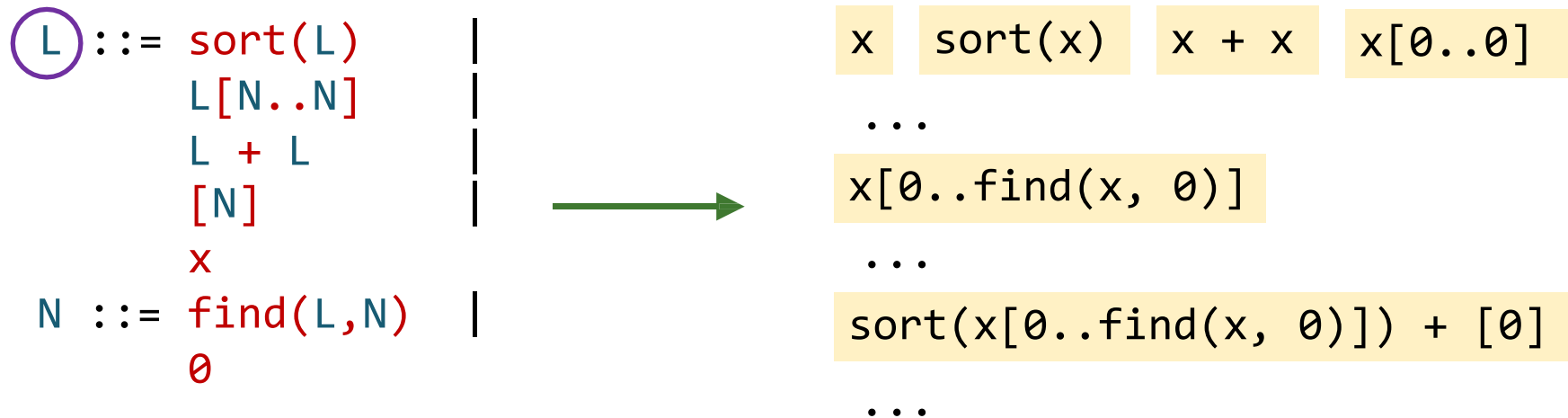
find(L+L,N)

find(x+x,0)

sort(L+L)

CFGs as structural constraints

Space of programs
=
all **ground**, **whole** programs

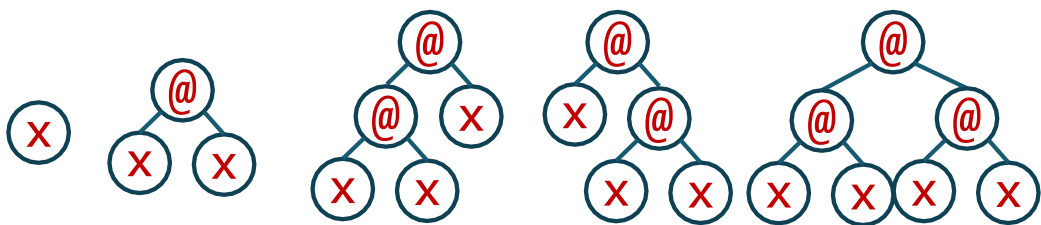


How big is the space?

$E ::= x \mid E @ E$

depth ≤ 0  $N(0) = 1$

depth ≤ 1  $N(1) = 2$

depth ≤ 2  $N(2) = 5$

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

How big is the space?

$E ::= x \mid E @ E$

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

$$N(d) \sim c^{2^d} \quad (c > 1)$$

$$N(1) = 1$$

$$N(2) = 2$$

$$N(3) = 5$$

$$N(4) = 26$$

$$N(5) = 677$$

$$N(6) = 458330$$

$$N(7) = 210066388901$$

$$N(8) = 44127887745906175987802$$

$$N(9) = 1947270476915296449559703445493848930452791205$$

$$N(10) = 3791862310265926082868235028027893277370233152247388584761734150717768254410341175325352026$$

How big is the space?

$$E ::= E \overset{x_1}{@_1} E \mid \dots \mid E \overset{x_k}{@_k} E$$

$$N(\emptyset) = k$$

$$N(d) = k + m * N(d - 1)^2$$

$$N(1) = 3$$

$$N(2) = 30$$

$$N(3) = 2703$$

$$N(4) = 21918630$$

$$N(5) = 1441279023230703$$

$$N(6) = 6231855668414547953818685622630$$

$$N(7) = 116508075215851596766492219468227024724121520304443212304350703$$

$$k = m = 3$$

The SyGuS project

[Alur et al. 2013]

<https://sygus.org/>

Goal: Unify different syntax-guided approaches

Collection of synthesis benchmarks + yearly competition

- 6 competitions since 2013
- consider writing a SyGuS solver for your project!

Common input format + supporting tools

- parser, baseline synthesizers

SyGuS problems

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, ...

\wedge , \vee , \neg , $+$, \leq , *ite*

CFG with terminals in the theory
(+ input variables)

Example: Conditional LIA
expressions w/o sums

$E ::= x \mid \textit{ite } C \ E \ E$

$C ::= E \leq E \mid C \wedge C \mid \neg C$

SyGuS problems

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



A first-order logic formula over
the theory



Examples:

$$f(0, 1) = 1 \wedge$$

$$f(1, 0) = 1 \wedge$$

$$f(1, 1) = 1 \wedge$$

$$f(2, 0) = 2$$

SyGuS problems

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

A first-order logic formula over
the theory

Examples:

$f(0, 1) = 1 \wedge$
 $f(1, 0) = 1 \wedge$
 $f(1, 1) = 1 \wedge$
 $f(2, 0) = 2$

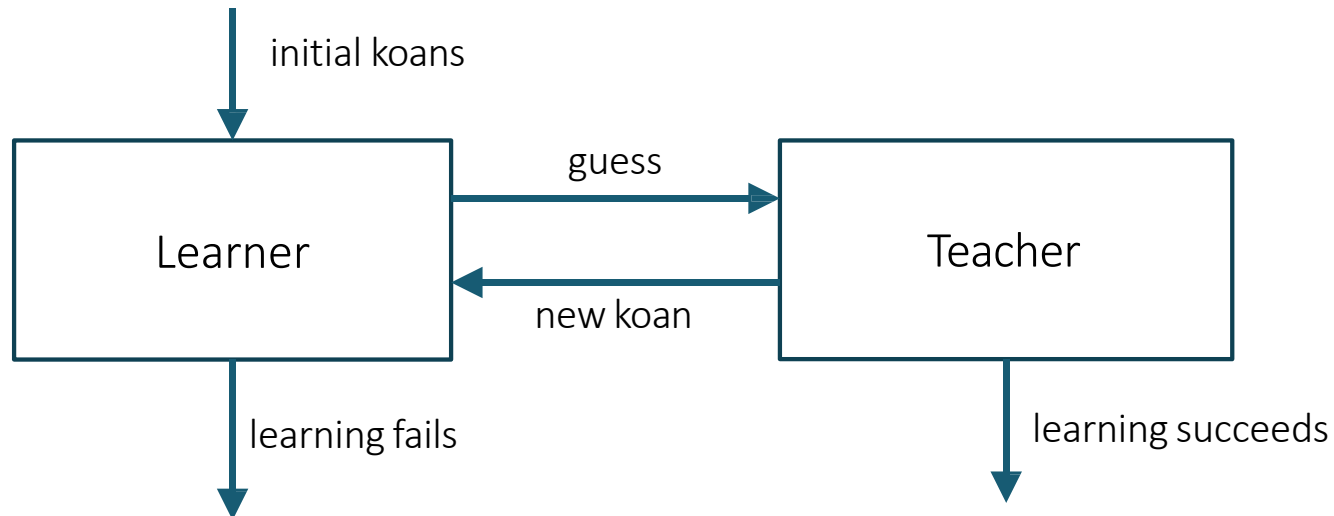
Formula with free variables:

$x \leq f(x, y) \wedge$
 $y \leq f(x, y) \wedge$
 $(f(x, y) = x \vee f(x, y) = y)$

can inductive synthesis
handle these?

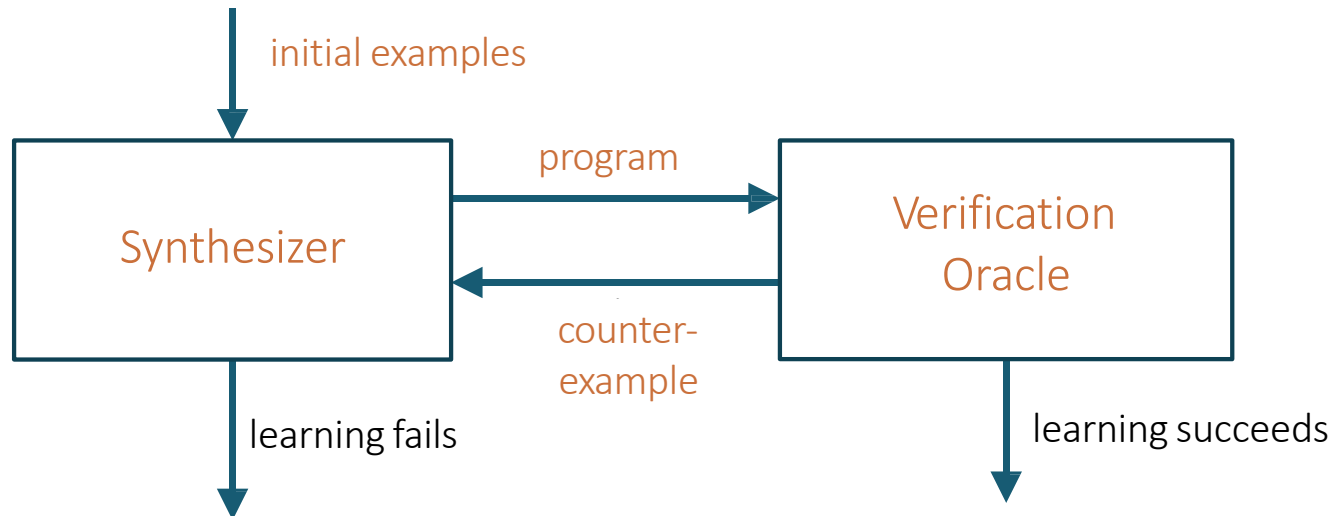
Counter-example guided inductive synthesis (CEGIS)

The Zendo of program synthesis

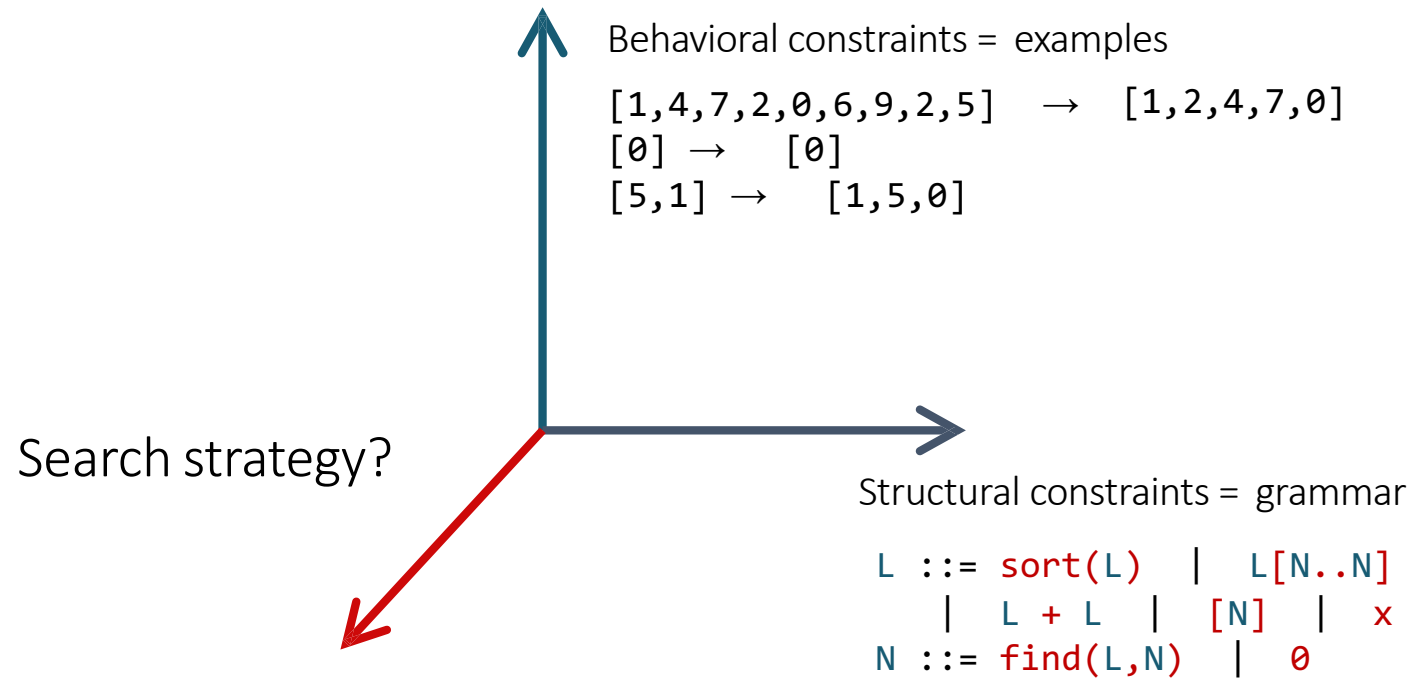


Counter-example guided inductive synthesis (CEGIS)

The Zendo of program synthesis



The problem statement



Enumerative search

Enumerative search

=

Explicit / Exhaustive Search

Idea: Sample programs from the grammar one by one and test them on the examples

Challenge: How do we systematically enumerate all programs?

bottom-up **vs** top-down

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: example

Program bank P

iter 0: `x` `0`

iter 1: `sort(x)` `x[0..0]` `x + x` `[0]`
`find(x,0)`

iter 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)`
`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)`

...

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

`[[1,4,0,6]]` \rightarrow `[1,4]`

Bottom-up enumeration

nonterminals rules (productions)
terminals starting nonterminal

```

bottom-up (<T, N, R, S>, [i → o])
{ P := [t | t in T && t is nullary]
  while (true)
    forall (p in P)
      if (whole(p) && p([i]) = [o])
        return p;
    P += grow(P);
}

grow (P) {
  P' := []
  forall (A ::= rhs in R)
    P' += [rhs[B -> p] | p in P, B →* p]
  return P';
}
  
```

Q: "Run" bottom-up on the board with:

```

L ::= sort(L)      |
    L[N..N]        |
    L + L          |
    [N]             |
    x               |
N ::= find(L,N)     |
    0               |
[[1,4,0,6] → [1,4]]
  
```

Top-down enumeration

Start from the start non-terminal
Expand remaining non-terminals using
productions

$L ::= L[N..N] \quad |$

$N ::= \overset{x}{\text{find}}(L, N) \quad |$
 $\quad \quad \quad \emptyset$

$[[1, 4, \emptyset, 6] \rightarrow [1, 4]]$

Top-down: example

Worklist P

iter 0: L

iter 1: x[×] L[N..N]

iter 2: L[N..N]

iter 3: x[N..N] L[N..N][N..N]

iter 4: x[0..N] x[find(L,N)..N] L[N..N][N..N]

iter 5: x[0..0][×] x[0.. find(L,N)] x[find(L,N)..N] ...

iter 6: x[0.. find(L,N)] x[find(L,N)..N]

iter 7: x[0.. find(x,N)] x[0.. find(L[N..N],N)]

iter 8: x[0.. find(x,0)][✓] x[0.. find(x,find(L,N))]

iter 9:

L ::= L[N..N] | ←

x

N ::= find(L,N) | ←

0 ←

[[1,4,0,6] → [1,4]]

Top-down enumeration

```
top-down(<T, N, R, S>, [i →  
                                o]) {  
    P := [S]  
    while (P != [])  
        p := P.dequeue();  
        if (ground(p) && p([i]) = [o])  
            return p;  
        P.enqueue(unroll(p));  
}  
  
unroll(p) {  
    P' := []  
    forall (A in p)  
        forall (A ::= rhs in R)  
            P' += p[A -> rhs]  
    return P';  
}
```

Q: “Run” top-down on the board with

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

$[[1,4,0,6] \rightarrow [1,4]]$

Bottom-up vs top-down

Bottom-up

Top-down

Smaller to larger

- Has to explore between $3 \cdot 10^9$ and 10^{23} programs to find
`sort(x[0..find(x, 0)]) + [0]` (depth 6)

Candidates are **ground** but might not be **whole**

- Can always run on inputs
- Cannot always relate to outputs

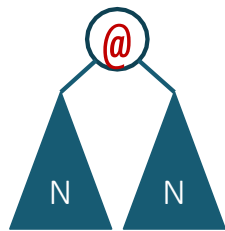
Candidates are **whole** but might not be **ground**

- Cannot always run on inputs
- Can always relate to outputs

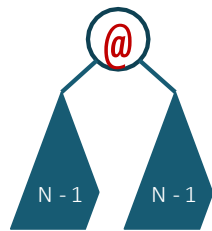
How to make it scale

Prune

Discard useless subprograms



$$m * N^2$$



$$m * (N - 1)^2$$

Prioritize

Explore more promising candidates first

$$P = \{ \begin{array}{l} [0][N..N] \\ x[N..N] \\ \dots \end{array} , \quad \leftarrow \text{dequeue this first}$$