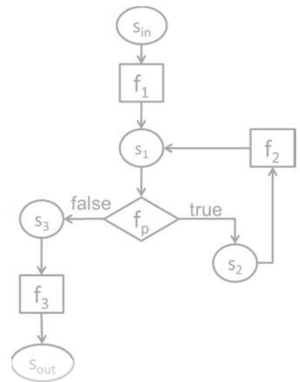
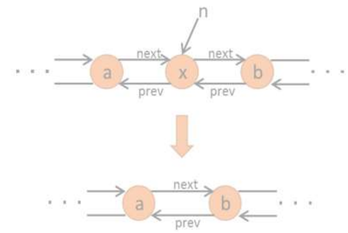
$$\exists c \forall in \; Q(c, in)$$

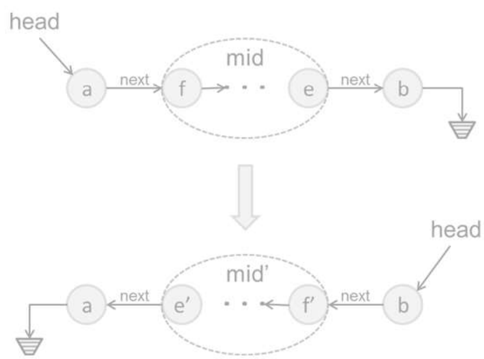
```
/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
  int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
  assert t == (x+y)/2;
  return t;
}
```

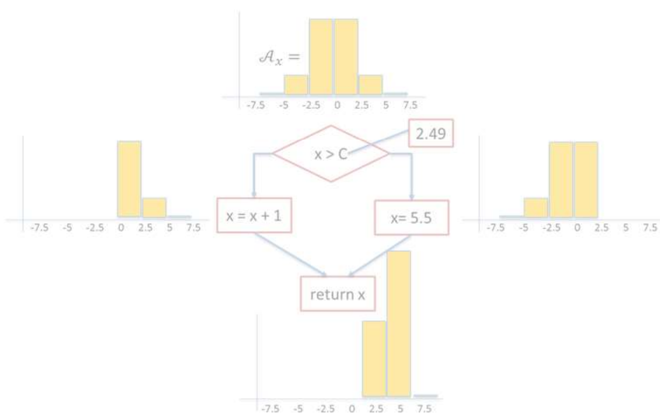





```
{
  s  = n.succ;
  p = n.pred;
  p.succ = s;
  s.pred = p;
}
```

Module I: Synthesizing Simple Programs





$$\varphi(p)$$

$$Sk[c](in)$$

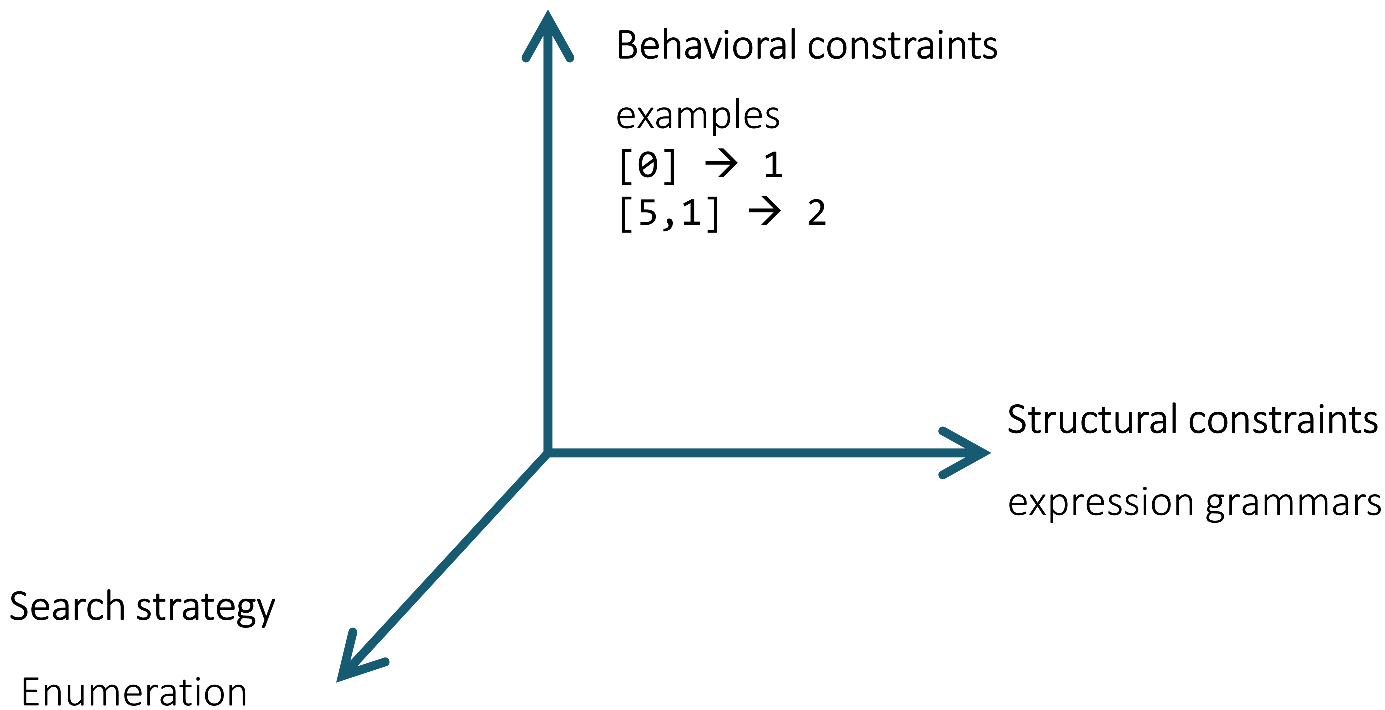
Lecture 2
Syntax-Guided Synthesis and Enumerative Search

Logistics

First review and team formation due next week

Other questions?

Week 1-2


Behavioral constraints
examples
[0] → 1
[5,1] → 2
Structural constraints
expression grammars
Search strategy
Enumeration

Today

Synthesis from examples

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 Programming

Inductive Learning

A little bit of history: inductive learning

MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970

Patrick Winston

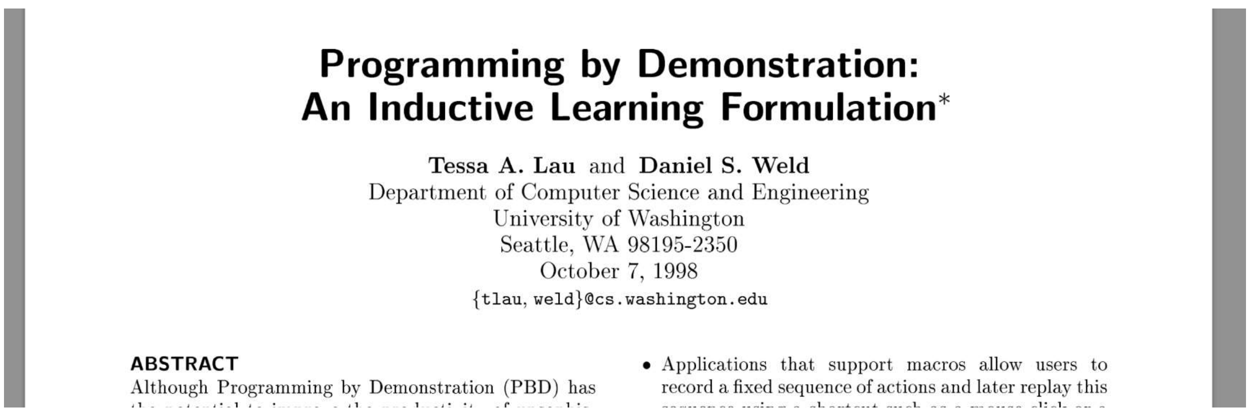
Explored the question of generalizing from a set of observations

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*

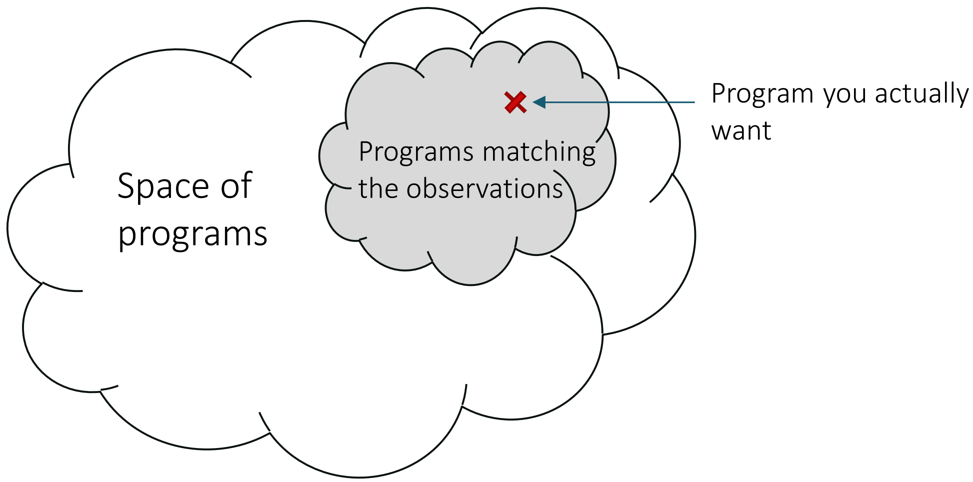
Tessa A. Lau and **Daniel S. Weld**
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
October 7, 1998
{tlau, weld}@cs.washington.edu

ABSTRACT
Although Programming by Demonstration (PBD) has

- Applications that support macros allow users to record a fixed sequence of actions and later replay this

Tessa Lau

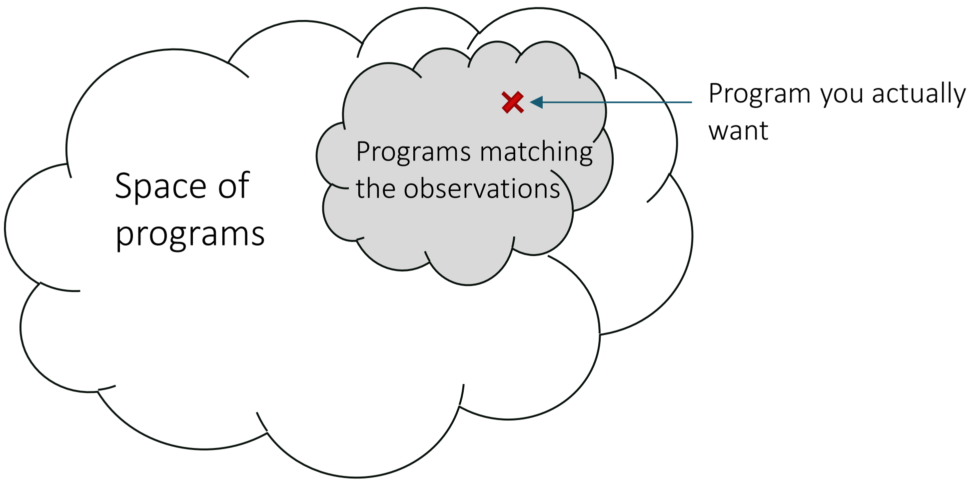
Key issues in inductive learning



(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



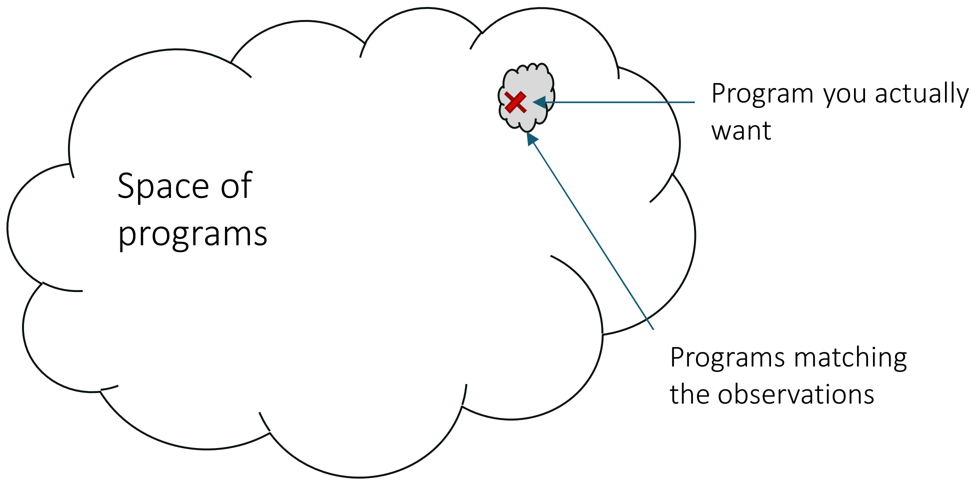
Traditional ML:

- Fix the space so that (1) is easy
- (2) becomes the main challenge

(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



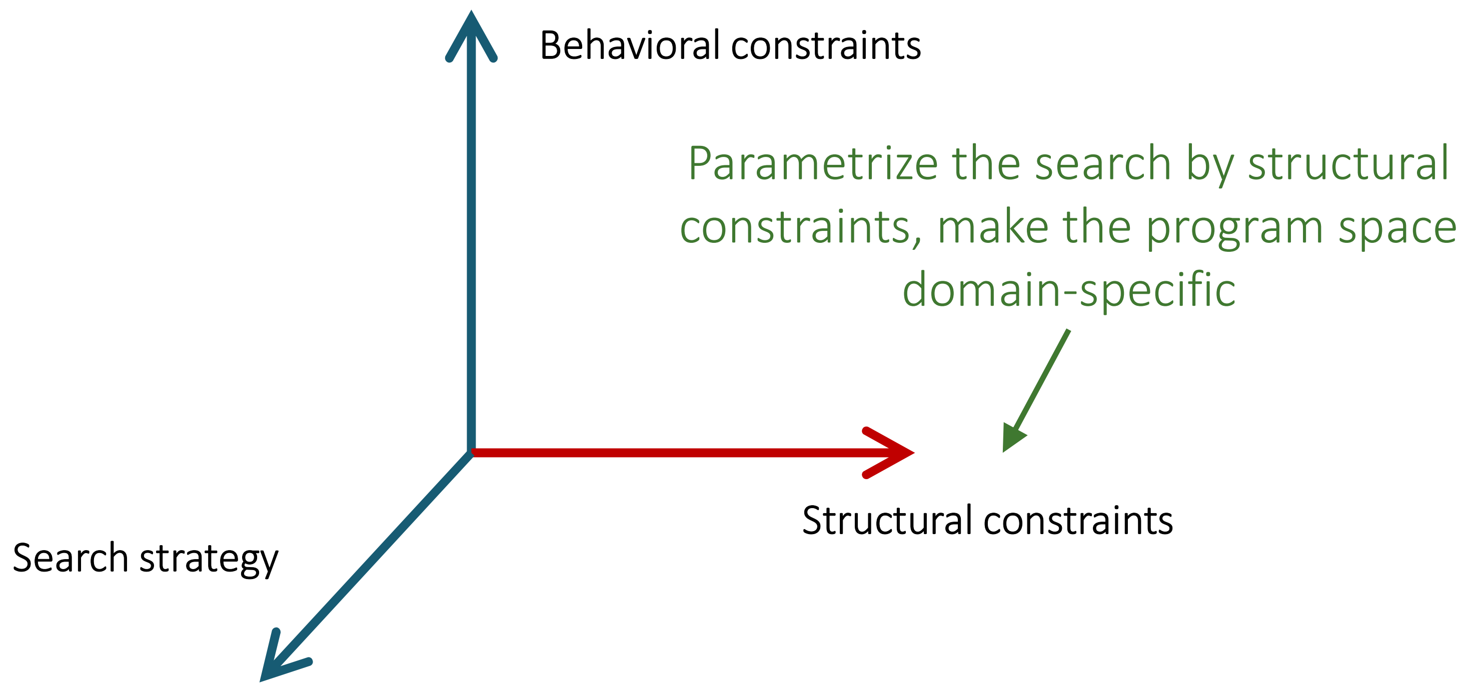
Program synthesis:

- Customize the space so that (2) becomes easier
- (1) is now the main challenge

(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


Behavioral constraints
Parametrize the search by structural constraints, make the program space domain-specific
Structural constraints
Search strategy

Syntax-Guided Synthesis

Example

```
[1,4,7,2,0,6,9,2,5,0]  →  [1,2,4,7,0]

L ::= x              |      the input
      single(N)      |      single(1) = [1]
      sort(L)        |      sort([6,9,2,5]) = [2,5,6,9]
      slice(L,N,N)   |      slice([6,9,2,5],0,2) = [6,9]
      concat(L,L)    |      concat([6,9],[2,5]) = [6,9,2,5]
N ::= find(L,N)      |      find([6,9],9) = 1
      0                     0

f(x) := concat( sort(slice(x,0,find(x,0))),  single(0))
```

Regular tree grammars (RTGs)

starting nonterminal

ranked alphabet (terminals)

nonterminals

```
L ::= x               |
      single(N)       |
      sort(L)         |
      slice(L,N,N)    |
      concat(L,L)     |
N ::= find(L,N)       |
      0
```

productions

Regular tree grammars (CFGs)

nonterminals   rules (productions)

alphabet   starting nonterminal

`<Σ, N, R, S>`

Terms: t $\in T_{\Sigma}(N)$ = all terms made from $N \cup \Sigma$

Rules are of the form: $A \to \sigma(A_1, \ldots, A_n)$

Derives in one step: $\mathcal{C}[A] \to \mathcal{C}[t]$ if $(A \to t) \in R$

(Incomplete) programs: $\{t \in T_{\Sigma}(N) | A \to^{*} t\}$

Ground programs: $\{t \in T_{\Sigma} | A \to^{*} t\}$

= programs without holes, complete programs

Whole programs: $\{t \in T_{\Sigma}(N) | S \to^{*} t\}$

= roughly, programs of the right type

```
concat(L,0)
```

$\mathrm{L} \to \mathrm{concat}(\mathrm{L}, \mathrm{L})$

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

find(concat(x,x),0)

sort(concat(L,L))
```

RTGs as structural constraints

Space of programs
= the *language* of an RTG
= all ground, whole programs

```
L ::= x              |
      single(N)      |
      sort(L)        |
      slice(L,N,N)   |
      concat(L,L)    |
N ::= find(L,N)      |
      0
```

→

`x` `sort(x)` `concat(x, x)` `slice(x,0,0)`

...

`slice(x,0,find(x,0))`

...

`concat(sort(slice(x,0,find(x,0))), single(0))`

...

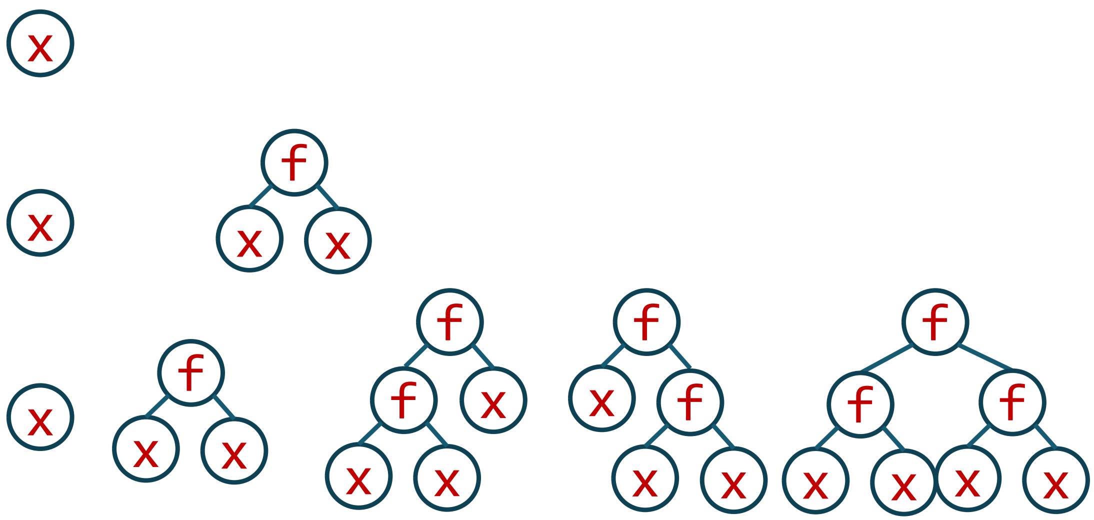
How big is the space?

`E ::= x | f(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 | f(E,E)
```

$$N(d) = 1 + N(d - 1)^2 \qquad N(d) \sim c^{2^d} \qquad (c > 1)$$

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 ::=     x_1 | ... | x_k |
 f_1(E,E) | ... | f_m(E,E)
```

$$E ::= x_1 \mid \ldots \mid x_k \mid f_1(E,E) \mid \ldots \mid f_m(E,E)$$

$$N(0) = k$$

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

N(1) = 3
N(2) = 30
N(3) = 2703
N(4) = 21918630
N(5) = 1441279023230703
N(6) = 6231855668414547953818685622630
N(7) = 116508075215851596766492219468227024724121520304443212304350703

k = m = 3

Syntactic sugar

Instead of this:

```
L ::= x            |
      single(N)    |
      sort(L)      |
      slice(L,N,N) |
      concat(L,L)  |
N ::= find(L,N)    |
      0
```

We will often write this:

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

- allow custom syntax for terminal symbols
- not an RTG strictly speaking, but you know what we mean…

Syntactic sugar

```
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

[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
- Our book has some code I can provide with this as well

SyGuS problems

SyGuS problem = < theory, spec, grammar >

A “library” of types and function symbols

Example: Linear Integer Arithmetic (LIA)

```
True, False
0,1,2,...
∧, ∨, ¬, +, ≤, ite
```

RTG 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
```

SyGuS problems

SyGuS problem = < theory, spec, grammar >

A first-order logic formula over the theory

Examples:

```
f(0, 1) = 1 ∧
f(1, 0) = 1 ∧
f(1, 1) = 1 ∧
f(2, 0) = 2
```

SyGuS demo

SyGuS problems

SyGuS problem = < theory, spec, grammar >

can inductive synthesis handle these?

A first-order logic formula over the theory

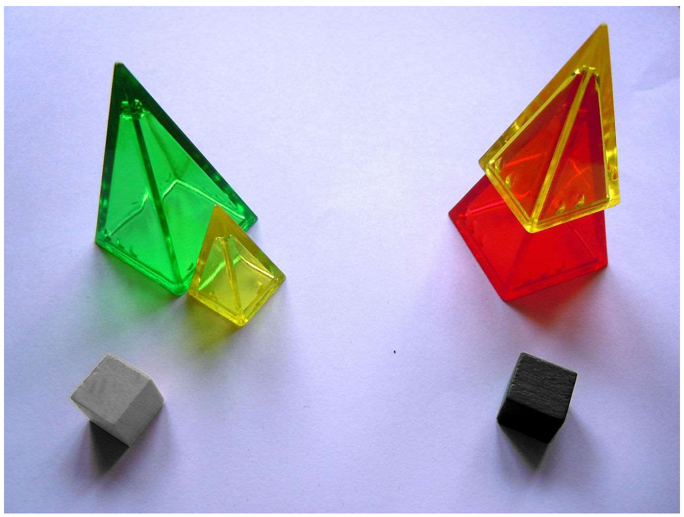
Examples:

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

Formula with free variables:

$$x \leq f(x, y) \land$$
$$y \leq f(x, y) \land$$
$$(f(x, y) = x \lor f(x, y) = y)$$

The Zendo game



The teacher makes up a secret rule

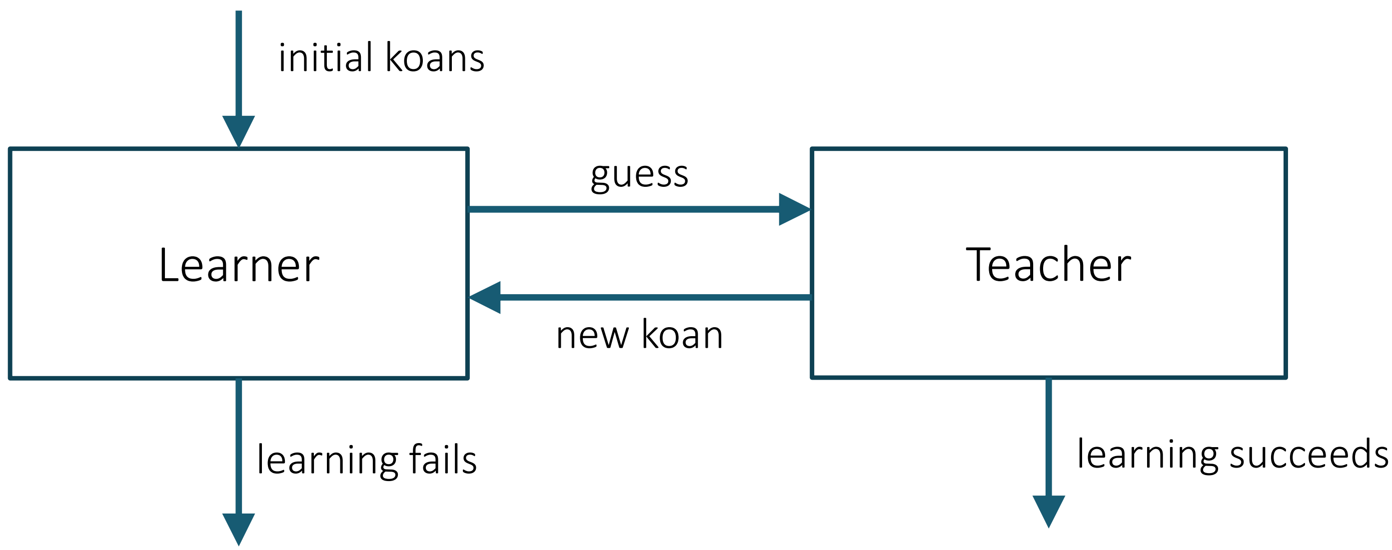
- e.g. all pieces must be grounded

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

A student can try to guess the rule

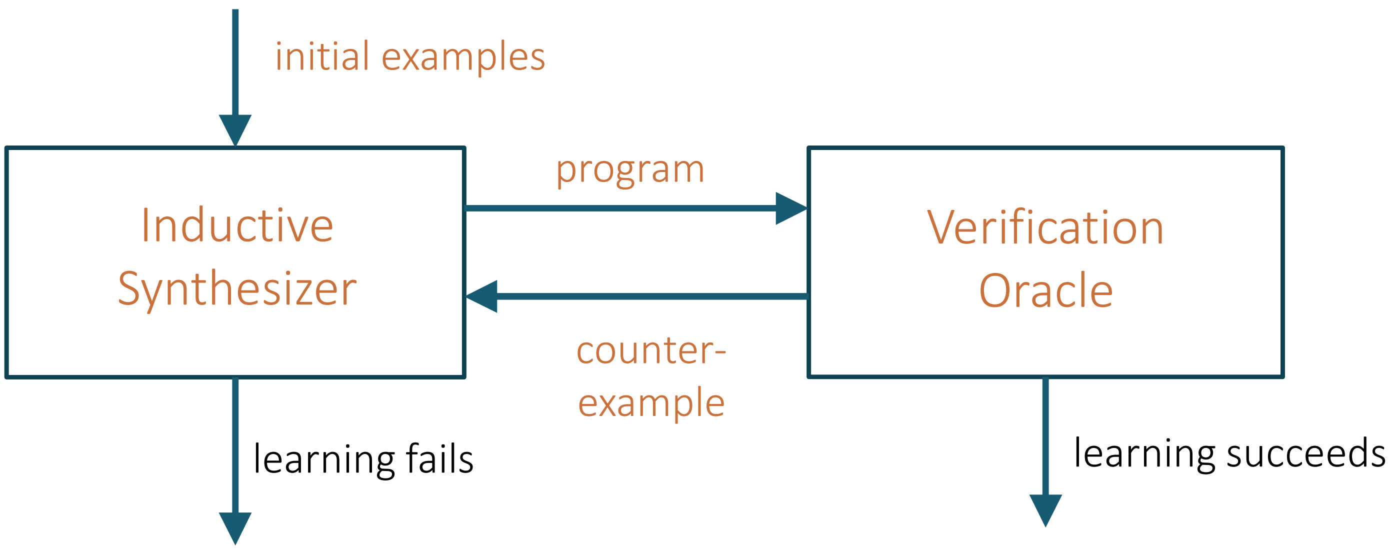
- if they are right, they win
- otherwise, the teacher builds a koan on which the two rules disagree

The Zendo game

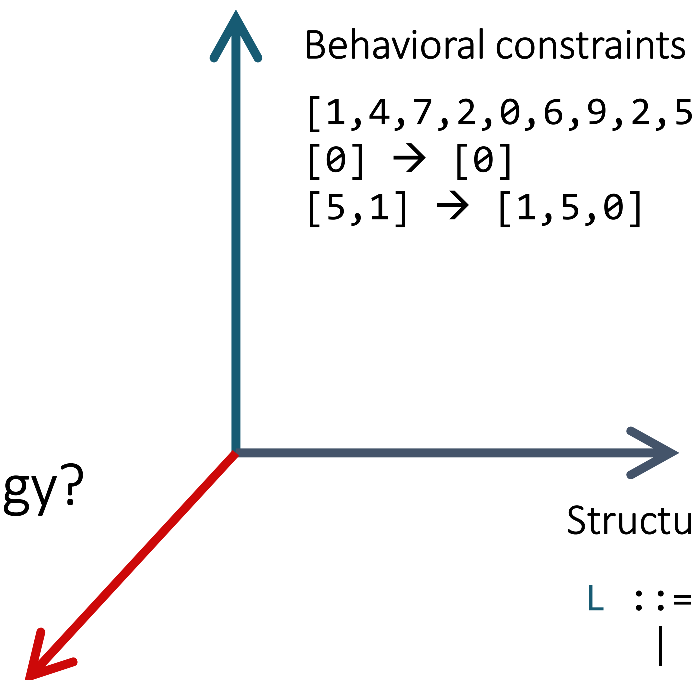


Counter-example guided inductive synthesis (CEGIS)

The Zendo of program synthesis



The problem statement



Behavioral constraints = examples

```
[1,4,7,2,0,6,9,2,5]  →  [1,2,4,7,0]
[0] → [0]
[5,1] → [1,5,0]
```

Search strategy?

Structural constraints = grammar

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

Enumerative search

Enumerative search

=

Explicit / Exhaustive Search

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

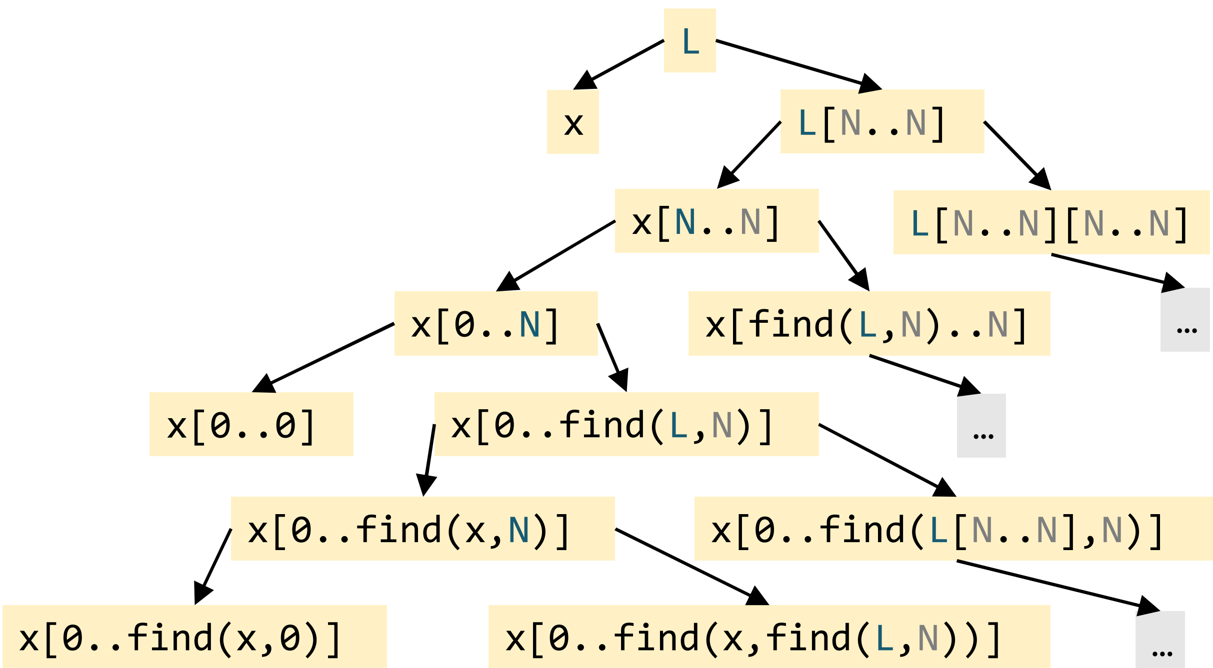
Challenge: How do we systematically enumerate all programs?

top-down vs bottom-up

Top-down enumeration: search space

Search space is a tree where

- nodes are incomplete programs
- edges are left-most "rewrites to"




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

```
[[1,4,0,6]  →  [1,4]]
```

Top-down enumeration = traversing the tree

Search tree can be traversed:

- depth-first (for fixed max depth)
- breadth-first
- best-first

General algorithm:

- Maintain a worklist of incomplete programs
- Initialize with the start non-terminal
- Expand left-most non-terminal using all productions

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

[[1,4,0,6] → [1,4]]
```

Top-down: algorithm

nonterminals rules (productions)

alphabet starting nonterminal

```
top-down(<Σ, N, R, S>, [i → o]):
  wl := [S]
  while (wl != []):
    p := wl.dequeue()
    if (ground(p) ∧ p([i]) = [o]):
      return p
    wl.enqueue(unroll(p))

unroll(p):
  wl' := []
  A := left-most non-term in p
  forall (A → rhs) in R:
    p' = p[A -> rhs]
    if !exceeds_bound(p'): wl' += p'
  return wl';
```

can be smart about what to dequeue

depth- or breadth-first
depending on where you enqueue

can impose bounds on depth/size

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

[[1,4,0,6] → [1,4]]
```

Top-down: example (depth-first)

Worklist `wl`

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]]`

Bottom-up enumeration

The dynamic programming approach

Maintain a bank of ground programs

Combine programs in the bank 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: algorithm (take 1)

nonterminals rules (productions)
alphabet starting nonterminal

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank := {}
  for d in [0..]:
    forall (A → rhs) in R:
      forall p in new-terms(A→rhs, d, bank):
        if (A = S ∧ p([i]) = [o]):
          return p
        bank += p;

new-terms(A → σ(A_1…A_n), d, bank):
 if (d = 0 ∧ n = 0) yield σ
 else forall <p_1,…,p_n> in bank^n:
         if A_i ->* p_i: yield σ(p_1,…,p_n)
```

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

[[1,4,0,6] → [1,4]]

inefficient, better index bank by non-terminal!

Bottom-up: algorithm (take 2)

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank[A] := {} forall A
  for d in [0..]:
    forall (A → rhs) in R:
      forall p in new-terms(A→rhs, d, bank):
        if (A = S ∧ p([i]) = [o]):
          return p
        bank += p;

new-terms(A → σ(A₁…Aₙ), d, bank):
 if (d = 0 ∧ n = 0) yield σ
 else forall <p₁,…,pₙ> in bank[A₁] × … × bank[Aₙ]:
            yield σ(p₁,…,pₙ)
```

inefficient, generating same terms again and again!
better index bank by depth

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

[[1,4,0,6] → [1,4]]

Bottom-up enumeration

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank[A,d] := {} forall A, d
  for d in [0..]:
    forall (A → rhs) in R:
      forall p in new-terms(A→rhs, d, bank):
        if (A = S ∧ p([i]) = [o]):
          return p
        bank[A,d] += p;

new-terms(A → σ(A_1…A_n), d, bank):
 if (d = 0 ∧ n = 0) yield σ
 else forall <d_1,…,d_n> in [0..d-1]^n s.t. max(d_1,…,d_n) = d-1:
       forall <p_1,…,p_n> in bank[A_1,d_1] × … × bank[A_n,d_n]:
         yield σ(p_1,…,p_n)
```

```
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

d= 0: `x` `0`

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

d = 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)` `sort([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)` `x[0..find(x,0)]`

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

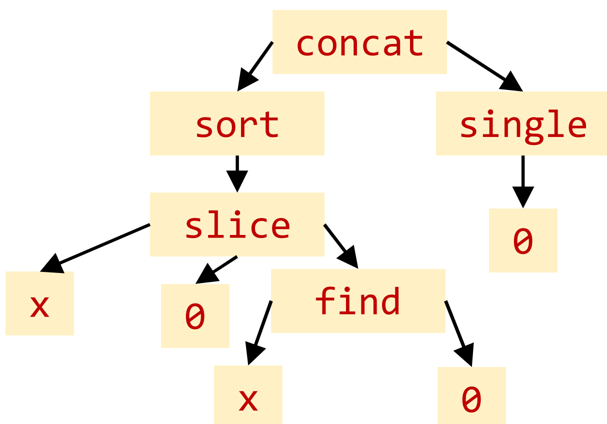
[[1,4,0,6] → [1,4]]

Bottom-up: discussion

What are some optimizations that come to mind?

Instead of by depth, we can enumerate by size

- Why would we want that?



depth = 4, size = 10

programs of size <= 10: 8667
programs of depth <= 4: >1M

- Which parts of the algo would we need to change?

Bottom-up vs top-down

Top-down | Bottom-up

Smaller to larger depth

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

Top-down

Candidates are whole but might not be ground

- Cannot always run on inputs
- Can always relate to outputs (?)

Bottom-up

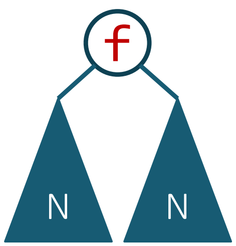
Candidates are ground but might not be whole

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

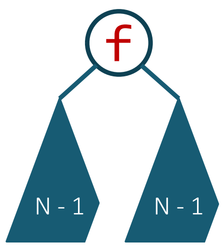
How to make it scale

Prune

Discard useless subprograms



$m * N^2$



$m * (N - 1)^2$

Prioritize

Explore more promising candidates first

```
P = { [0][N..N] ,
      x[N..N]   ,
      ... }
```

dequeue this first