CS5733 Program Synthesis

#2. <u>Sy</u>ntax-<u>Gu</u>ided <u>S</u>ynthesis

Ashish Mishra

Logistics

- Google Classroom
  - Has everyone joined the Classroom
  - Course webpage: Experimental, so please get the updates on the Classroom.
- Office Hours:
  - Monday: 3:30 - 4:30 pm.
- Project list up on the course page by this weekend
- Other questions?

Demo: Synquid: synthesis goal and components

Step 1: define synthesis goal as a *type*

sorted list

```
intersect :: xs:List a   →   ys:List a  →
                    List a
```

the set of elements

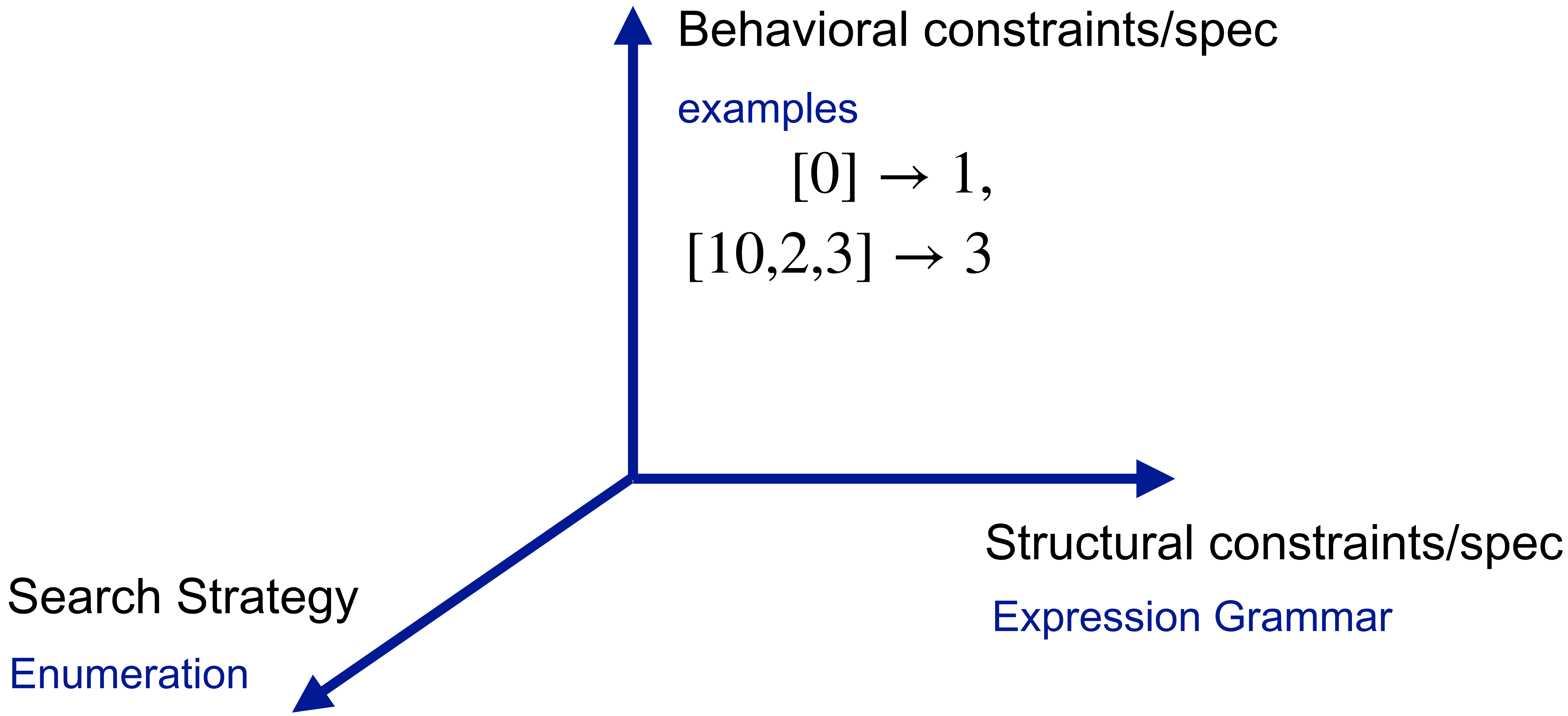
Step 2: define a set of components

- Which primitive operations is our function likely to use?
- Here: {`Nil`, `Cons`, `<`}

A demonstration of Synquid

- http://comcom.csail.mit.edu/demos/#intersection

Next two classes



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

=

Programming by Example /

Programming by Demonstration

=

Inductive Programming /

Inductive Learning /

Inductive Program Synthesis

Inductive learning: History

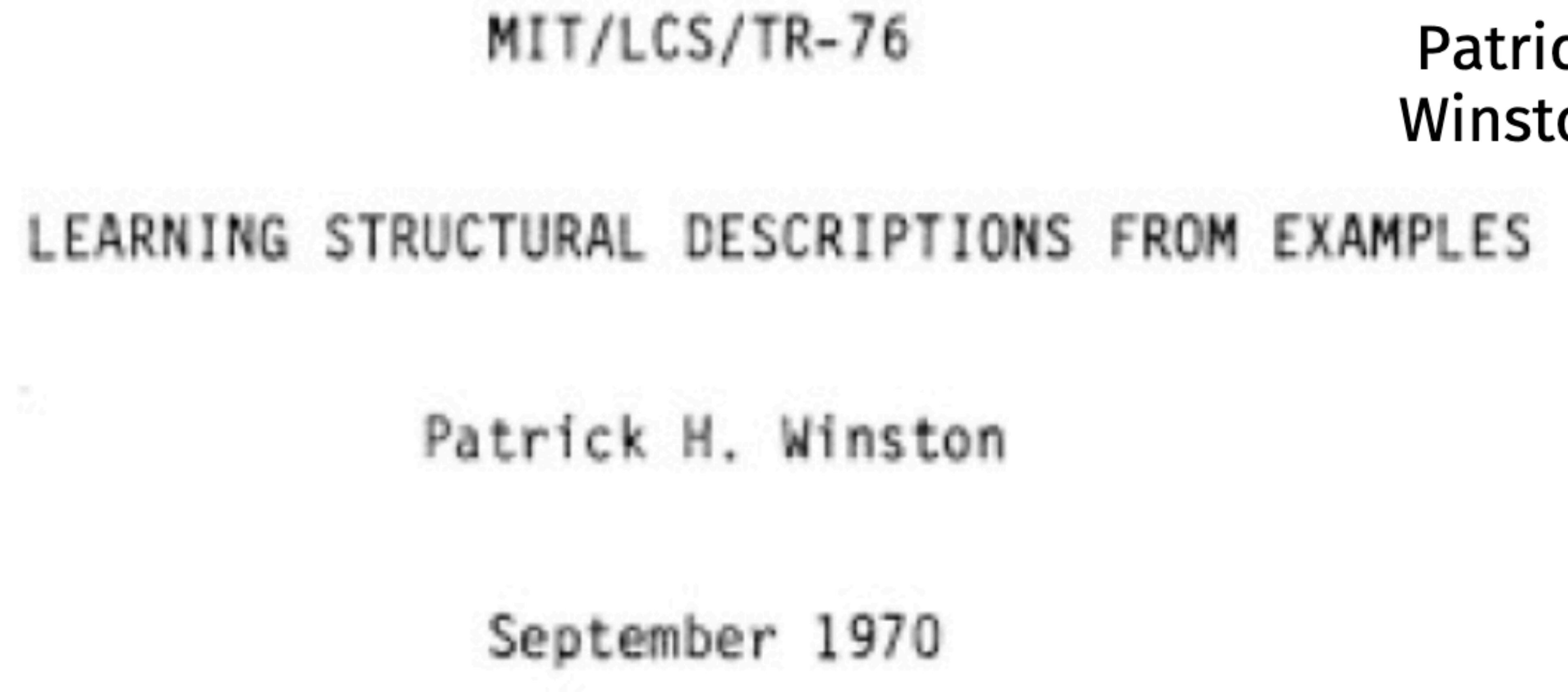
Patrick Winston

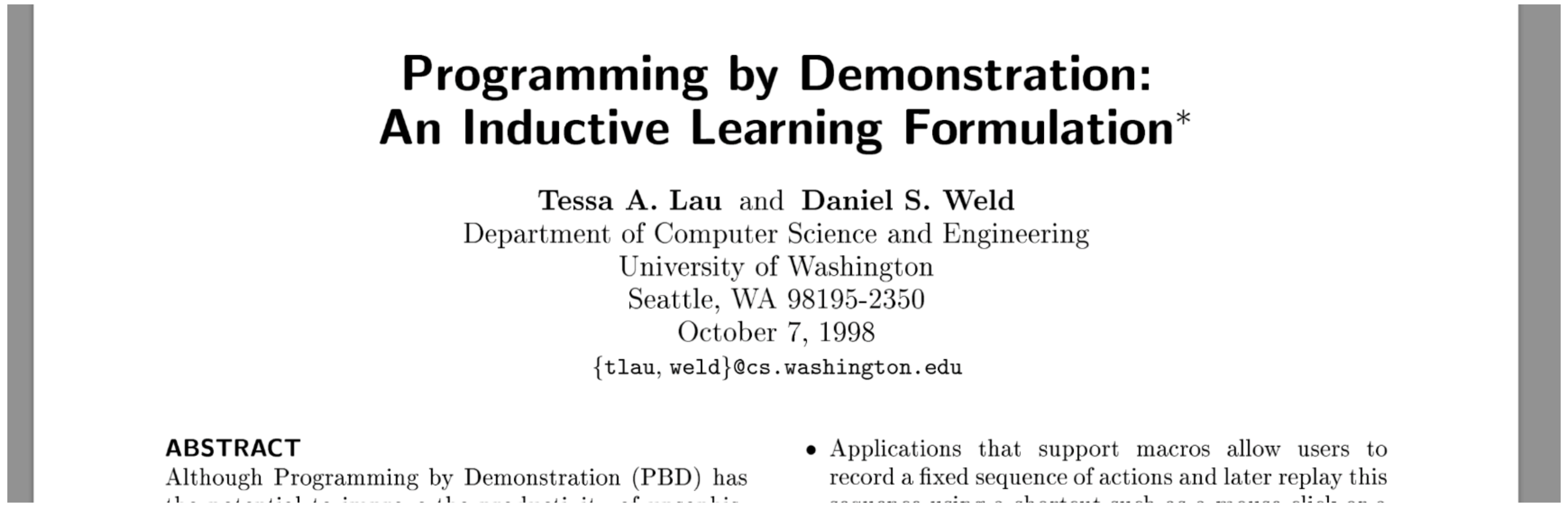
MIT/LCS/TR-76

LEARNING STRUCTURAL DESCRIPTIONS FROM EXAMPLES

Patrick H. Winston

September 1970





Programming by Demonstration: An Inductive Learning Formulation*

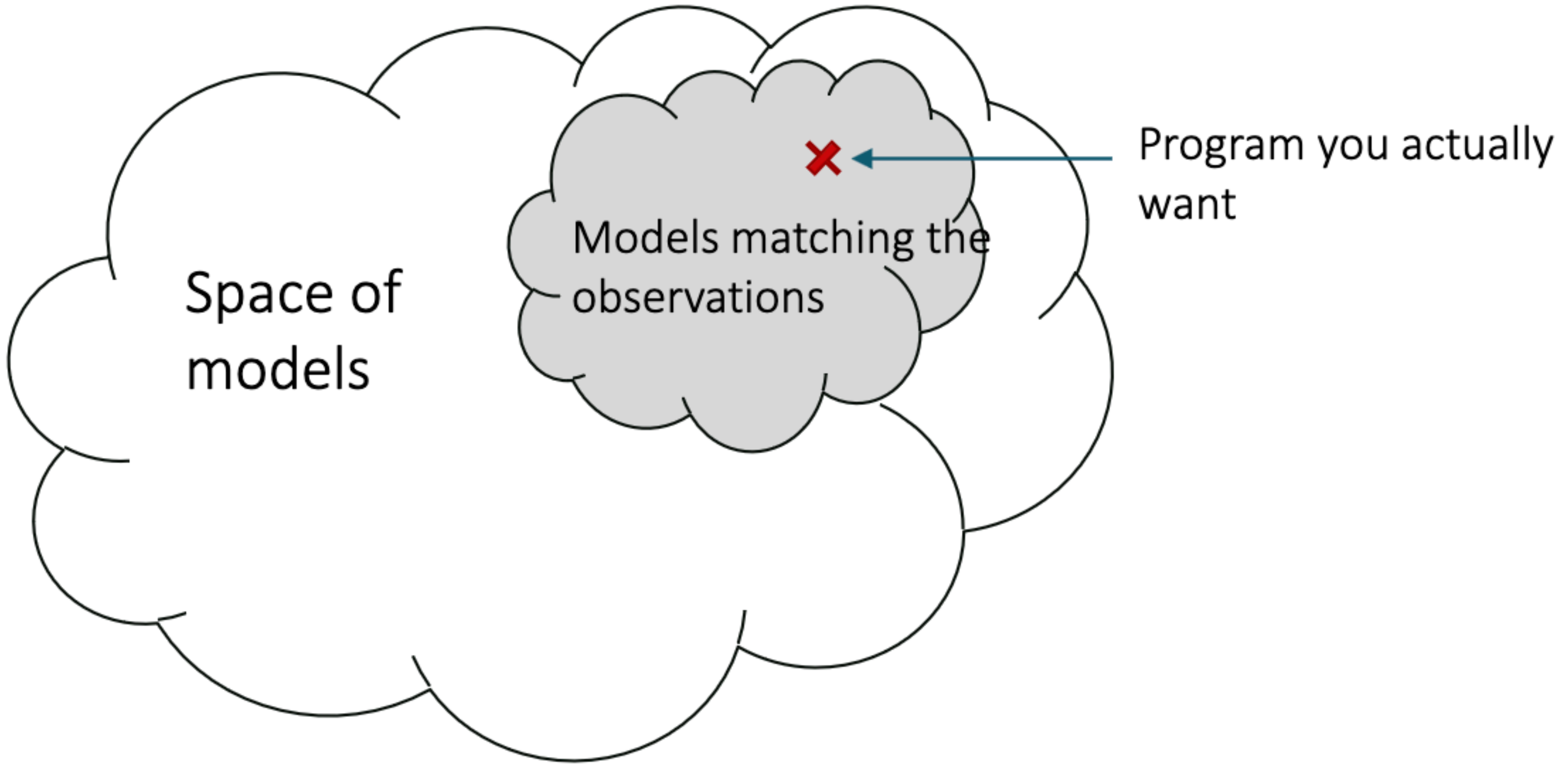
Tessa A. Lau and Daniel S. Weld
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195-2350
October 7, 1998
{tlau, weld}@cs.washington.edu

ABSTRACT
Although Programming by Demonstration (PBD) has

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

- Explored the question of generalizing from a set of observations.
- Became the foundation of machine learning.
- Lau's work aimed to develop general techniques that could be adapted to a variety of PBE problems.

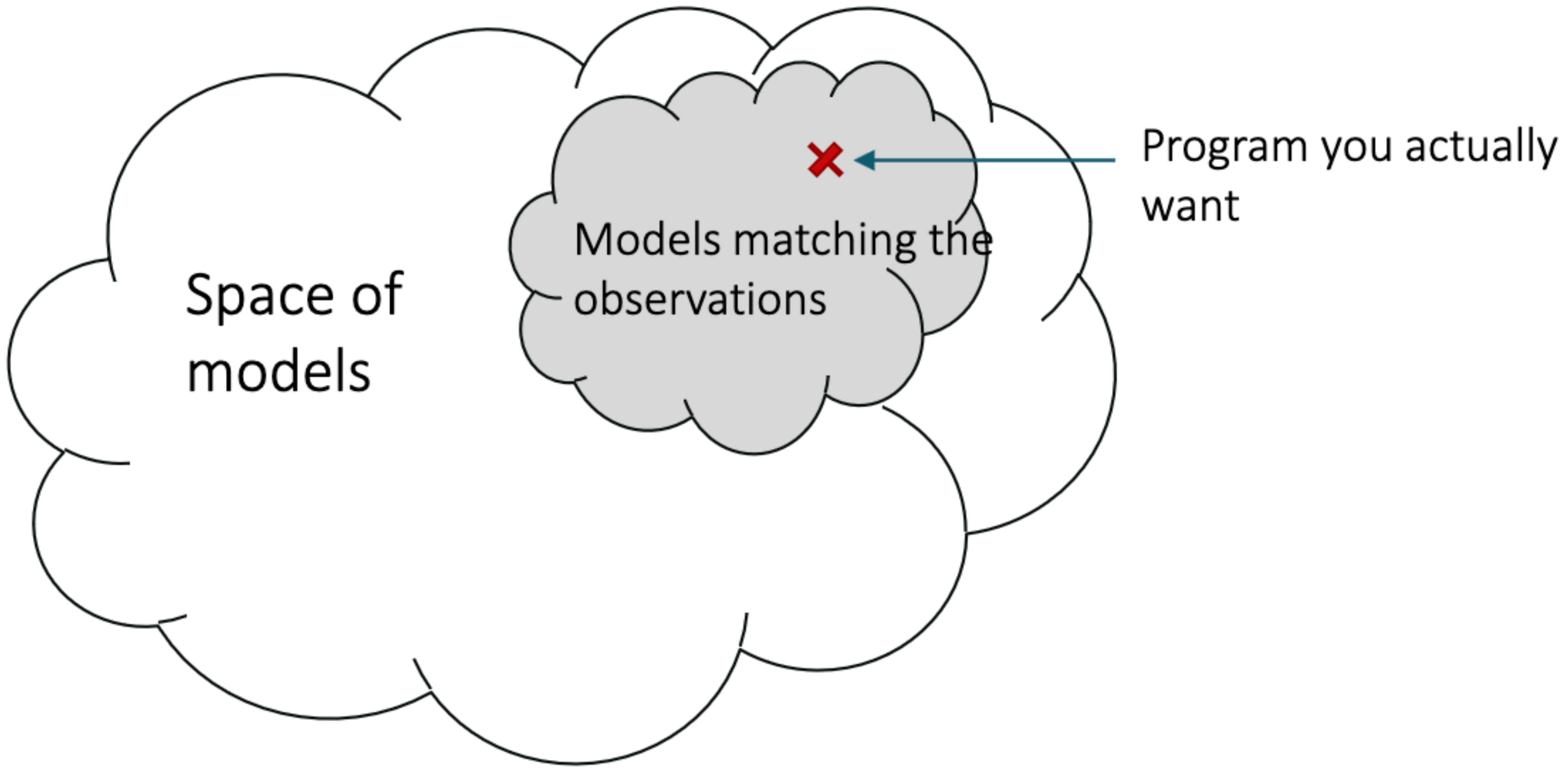
Key issues in inductive learning



1. How do you find a model that matches the observations?
2. How do you know it is the model you are looking for?

Inductive problems are mostly underspecified.

Key issues in inductive learning



Traditional ML:

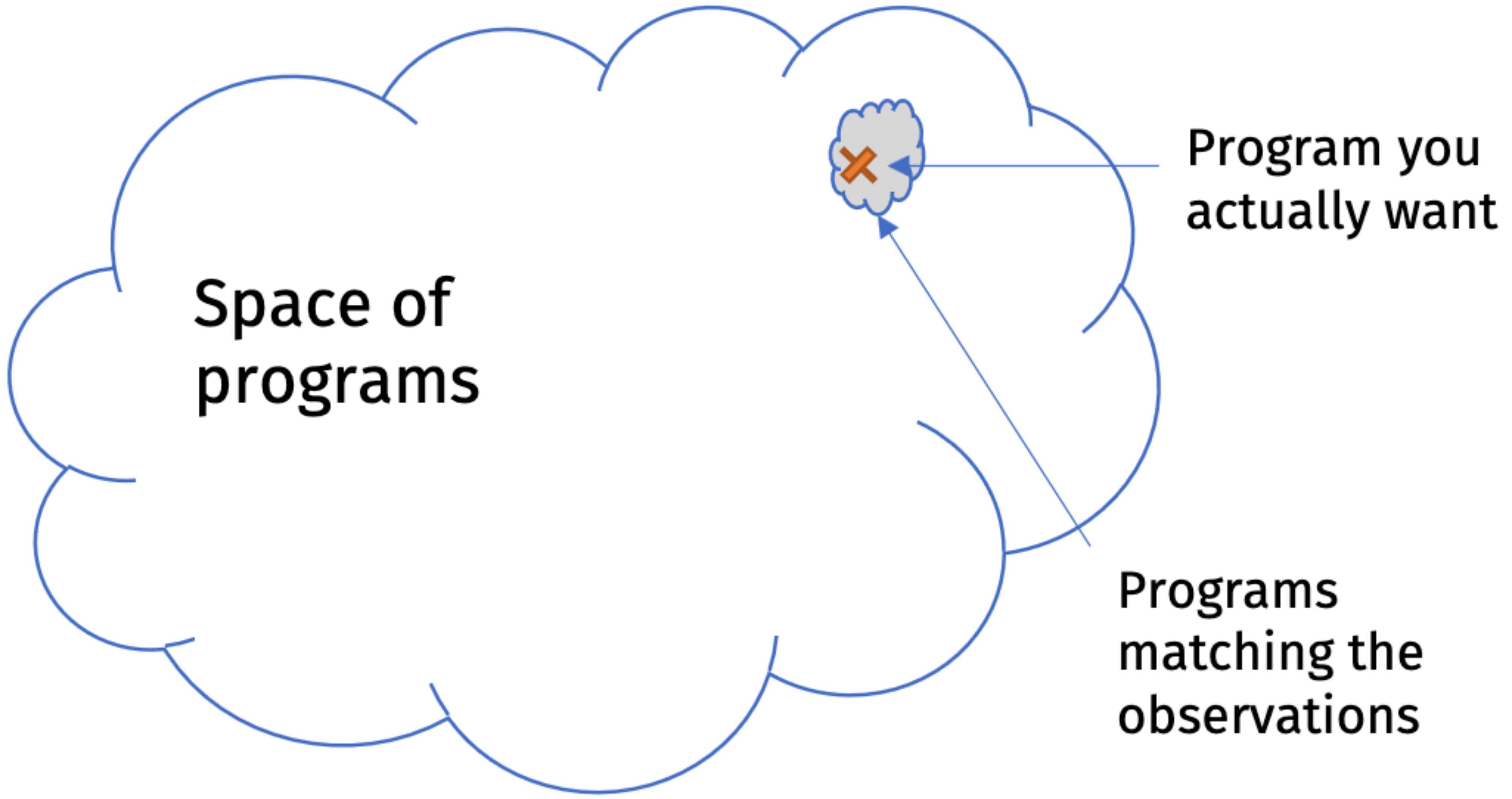
• Fix the space so that (1) is easy

- Pick extremely expressive programs (e.g. NN).
- Pick a space that is too restricted (e.g. SVMs).

• (2) becomes the main challenge.

1. How do you find a model that matches the observations?
2. How do you know it is the model you are looking for?

The Synthesis Approach

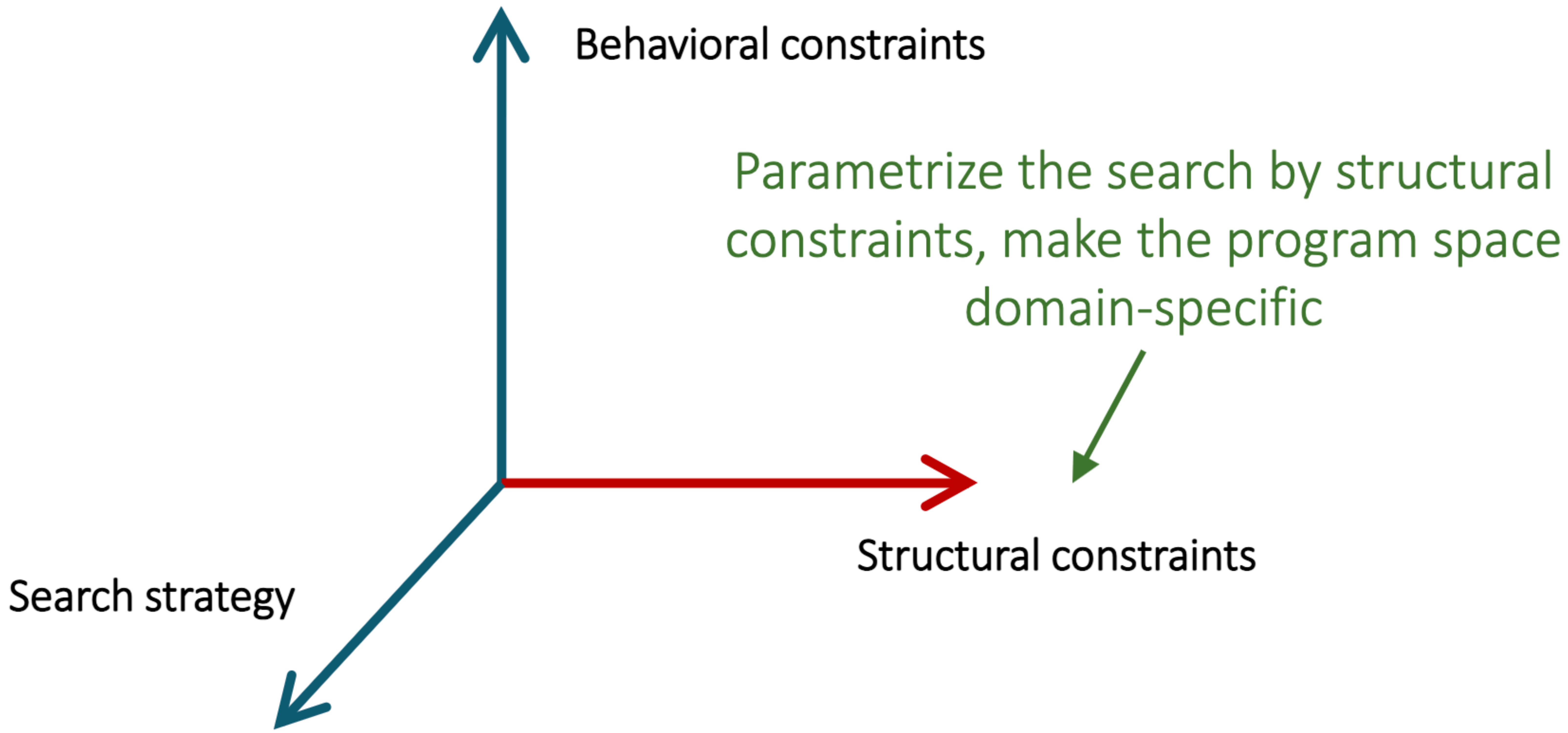


Program synthesis:

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

1. How do you find a model that matches the observations?
2. How do you know it is the model 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 (RTGs)

nonterminals rules (productions)
alphabet starting nonterminal

<Σ, N, R, S>

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

Derives in one step: $\mathcal{C}[A] \to \mathcal{C}[t]$ if $(A \to t) \in R$
A is the leftmost non-terminal in $\mathcal{C}[A]$

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

Complete terms/programs: $\{t \in T_\Sigma | A \to^* t\}$
= programs without holes

Whole programs: $\{t \in T_\Sigma | S \to^* t\}$
= roughly, programs of the right type

$L \to concat(L, L)$

```
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 $L(G)$
= all complete, 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))
...
```

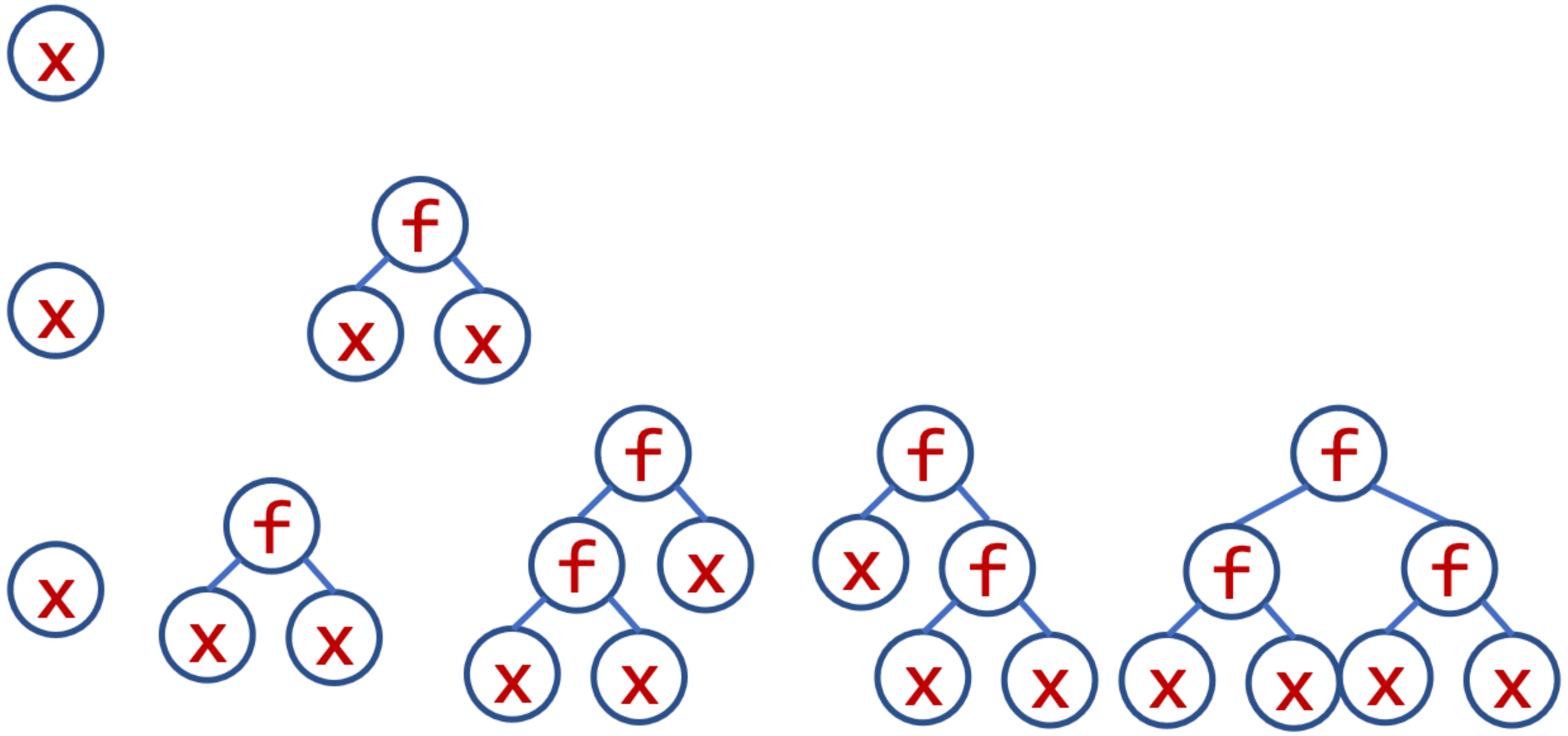
How big is the space?

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

depth <= 0

depth <= 1

depth <= 2



N(0) = 1

N(1) = 2

N(2) = 5

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

How big is the space?

$$E ::= x \mid f(E,E)$$

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

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 ::=     x₁ | ... | xₖ |
 f₁(E,E) | ... | fₘ(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

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

SyGuS Problems

SyGuS problem = < theory, spec, grammar >

- A Library of types and functions
- E.g. Theory of LIA

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

- RTG with Terminals in theory and i input variables
- Example: Conditional LIA expressions w/o sums

```
E ::= x | 0 | 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

- https://cvc5.github.io/app/#temp_17496bf7-0f49-4631-94a2-a521831246c5
-

SyGuS Problems

SyGuS problem = < theory, spec, grammar >

A first-order logic formula over the theory

Examples:

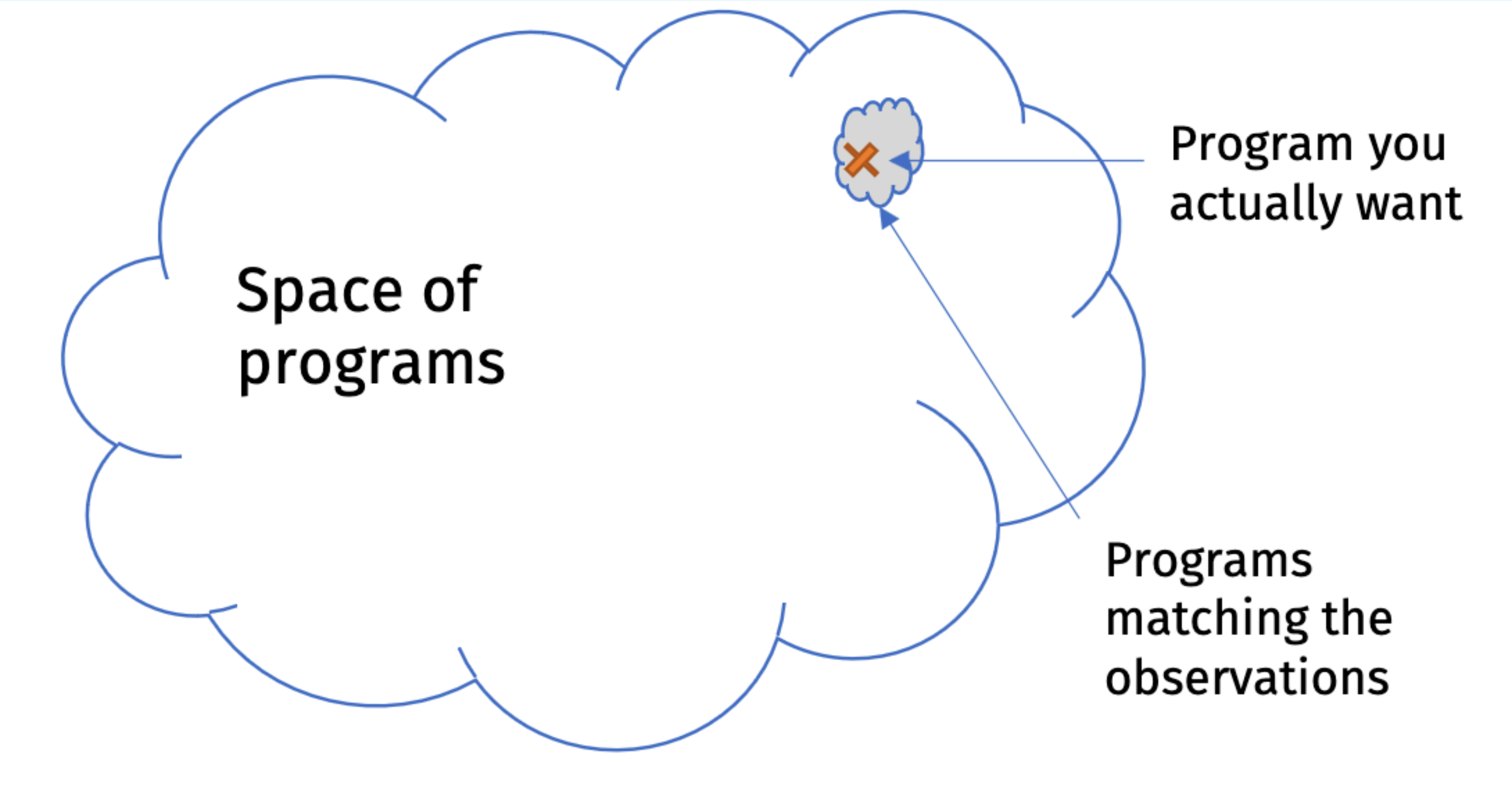
$$f(0, 1) = 1 \land$$
$$f(1, 0) = 1 \land$$
$$f(1, 1) = 1 \land$$
$$f(2, 0) = 2$$

Formula with free variables:

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

Can Inductive Synthesis Handle these?

Are syntax restrictions sufficient?



Space of Programs is still humongous

Can we do better?

Idea: dynamically Learn from Failures

Counter Example Guided Inductive Synthesis (CEGIS)

Teacher and Learner Model for Rule Discovery

Zendo (game)

文A **Add languages** ⌄

Article | Talk | Read | Edit | View history | Tools ⌄

From Wikipedia, the free encyclopedia

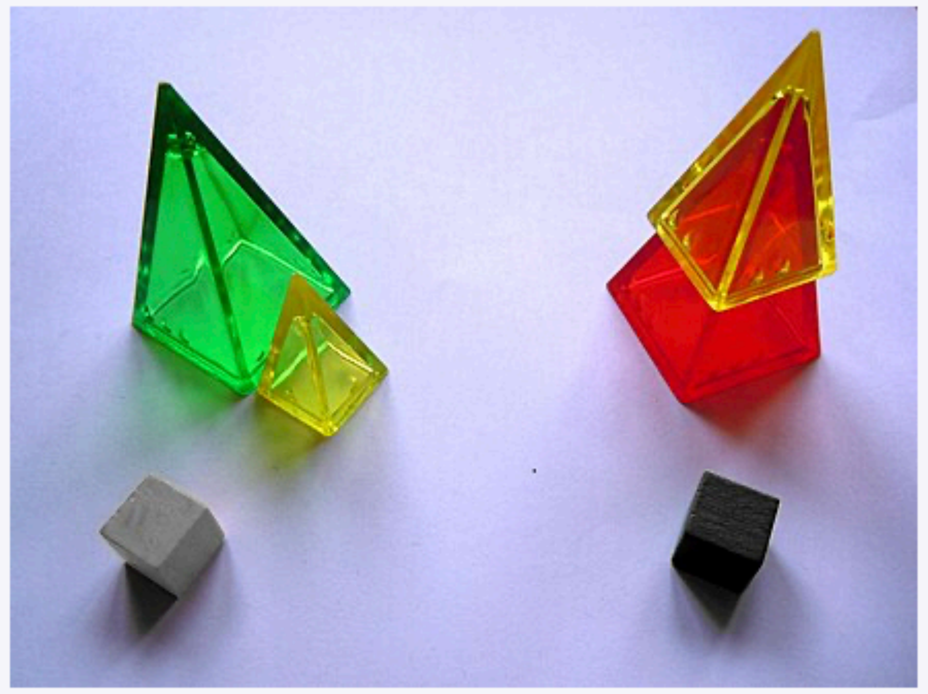
Zendo is a game of inductive logic designed by Kory Heath in which one player (the "Master") creates a rule for structures ("koans") to follow, and the other players (the "Students") try to discover it by building and studying various koans which follow or break the rule. The first student to correctly state the rule wins.

Zendo can be compared to the card game *Eleusis* and the chess variant *Penultima* in which players attempt to discover inductively a secret rule thought of by one or more players (called "God" or "Nature" in *Eleusis* and "Spectators" in *Penultima*) who declare plays legal or illegal on the basis of their rules. It can also be compared to Petals Around the Rose, a similar inductive reasoning puzzle where the "secret rule" is always the same.

The game can be played with any set of colorful playing pieces, and has been sold with a set of 60 Icehouse pyramids in red, yellow, green, and blue, 60 glass stones and a small deck of cards containing simple rules for beginners. The Icehouse pieces were replaced in the second edition with blocks, single size pyramids and wedges. Origami pyramids are a common choice of playing piece.

Zendo

The Game of Inductive Logic



The beginning of a game of *Zendo*. According to the marker stones, the koan on the left follows the Master's rule, but the one on the right does not.

Designers	Kory Heath
Publishers	Looney Labs
Publication	December 31, 1999; 24 years ago

Teacher and Learner Model for Rule Discovery

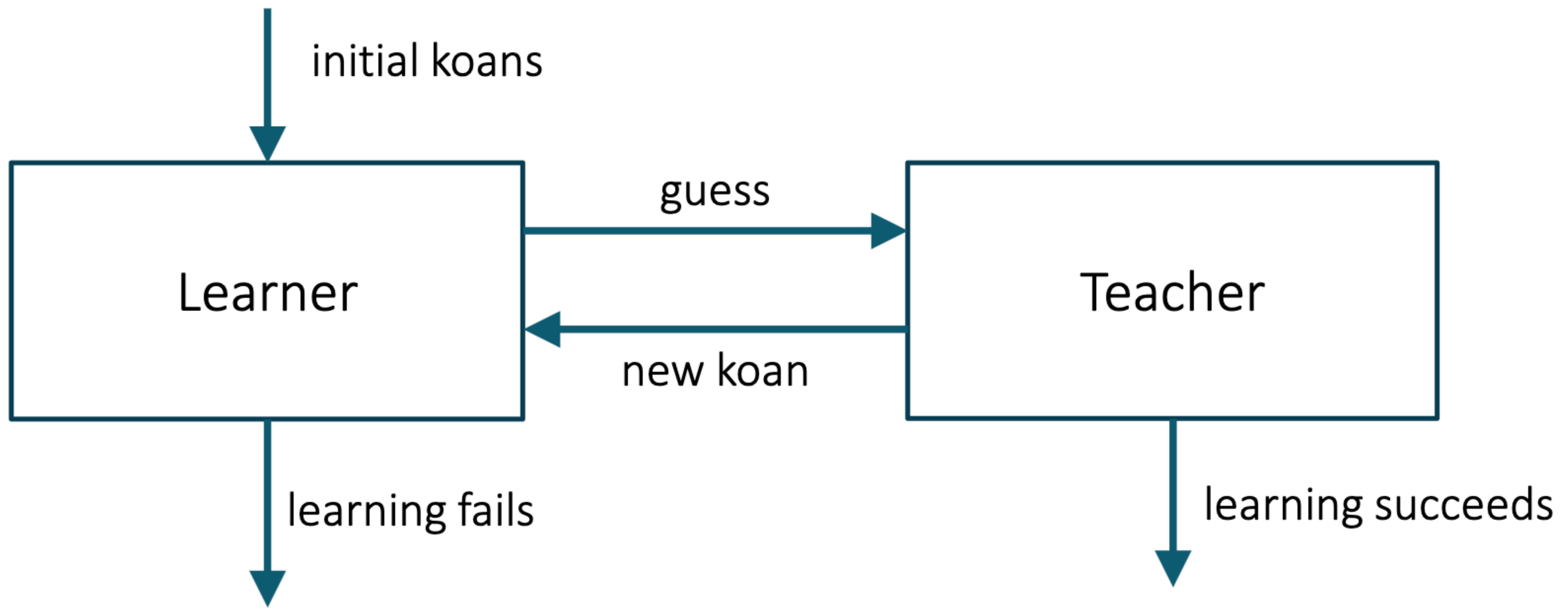
The teacher makes up a secret rule

- e.g. all pieces must be grounded

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

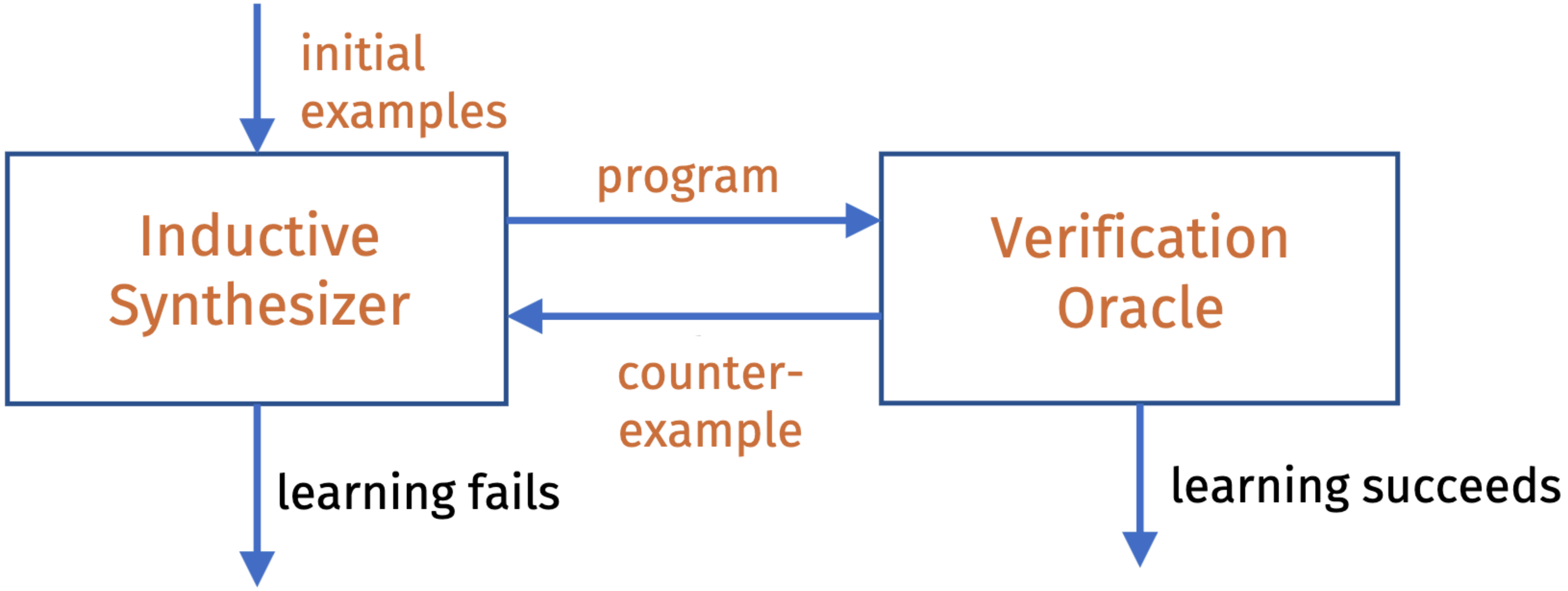
A student can try to guess the rule

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



Counter-example guided inductive synthesis (CEGIS)

The Zendo of program synthesis



The duality bw Verification and Synthesis

Example: CEGIS

Spec for max function

$$\max(x,y) \geq x \land \max(x,y) \geq y \land$$
$$(\max(x, y) = x \lor \max(x, y) = y)$$

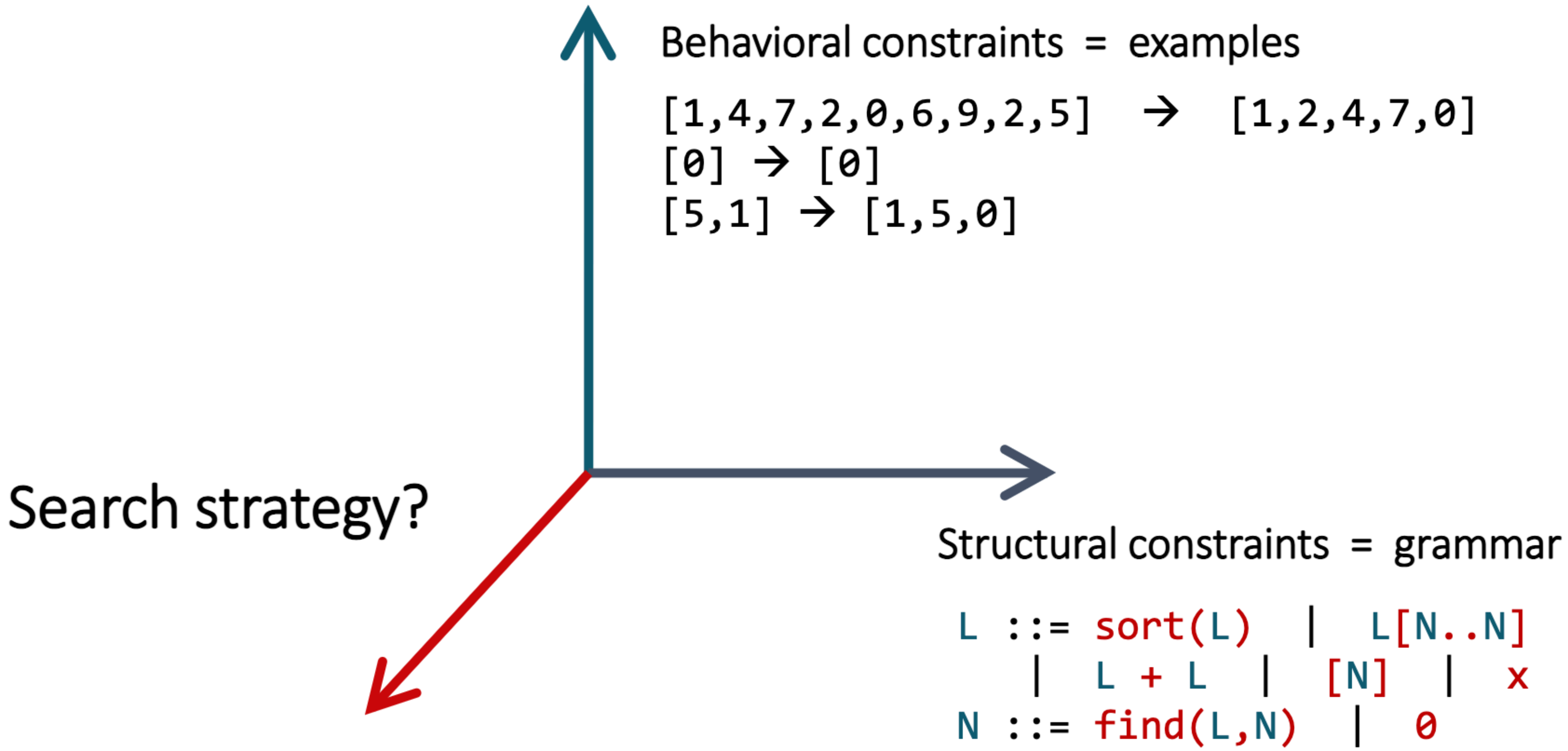
Program Space

```
expr = expr + expr | expr - expr | x | y  |
          | 0  | 1 | ITE (bexp, expr, expr)
bexp = expr relop expr
relop = .....
```

Expr	Counter Example
x	<x = 0, y = 1>
y	<x = 1, y = 0>
1	<x = 0, y =0>
x + y	<x = 1, y = 1>
ITE (x <= y, y, x)	_

The final dimension
Search Strategies

Revisiting the Problem



Enumerative Search

=

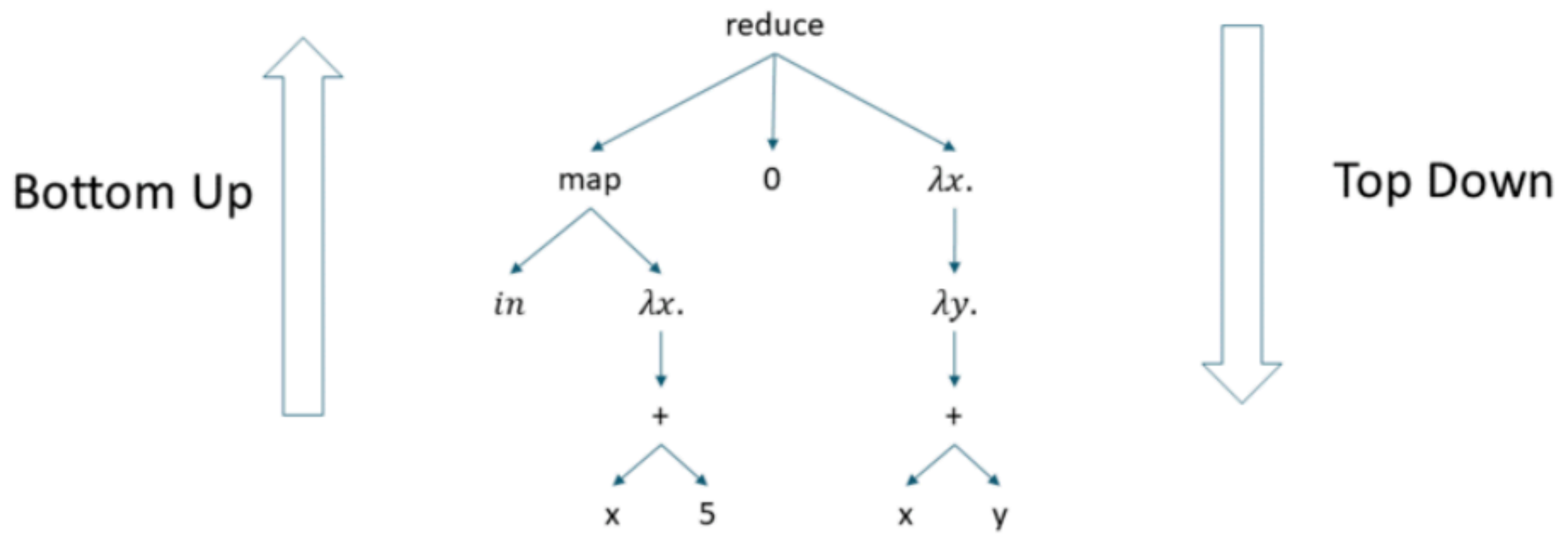
Explicit / Exhaustive Search

Idea: Enumerate 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

reduce (map $in\ \lambda$ x. x + 5) 0 $\lambda\, x.\, \lambda y.\, x + y$



FP Trivia:

reduce (map in $\lambda x\,.\,x + 5)\ 0\ (\lambda x\,.\,\lambda y\,.\,(x + y))$

functions : map, reduce

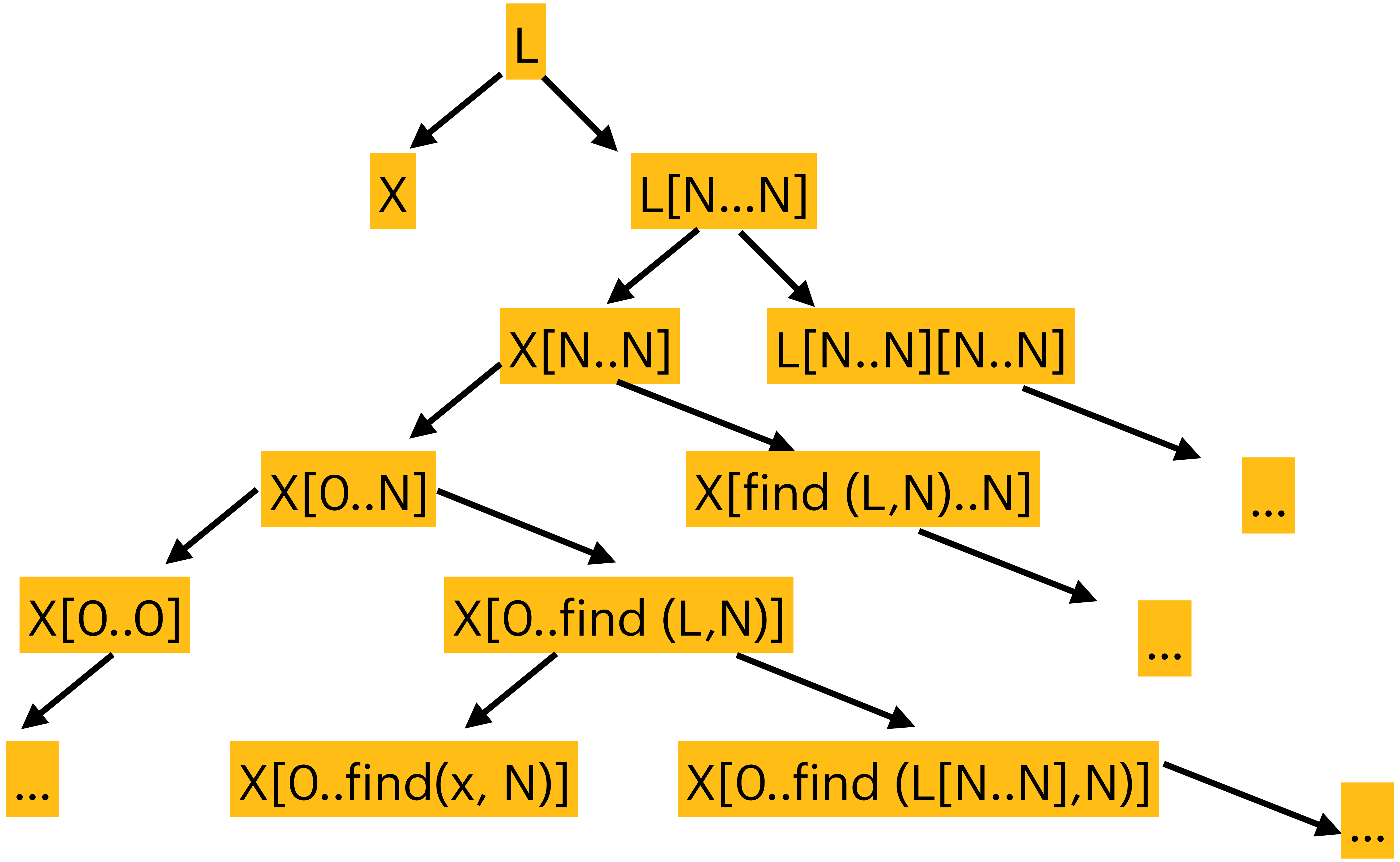
Top-down enumeration: search space

Search space is a tree where

- nodes are whole incomplete programs
- edges are “derives in one step”

```
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
  - later in class: 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]]
```