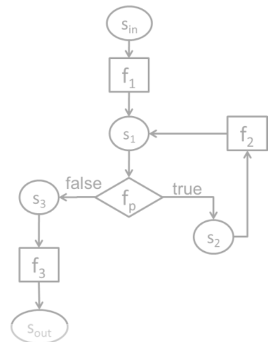
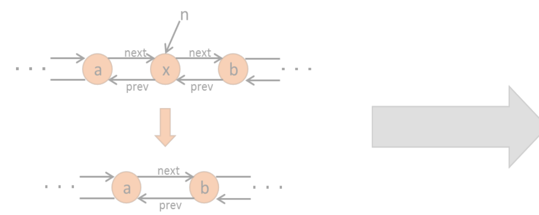
$\exists c \forall in\ Q(c, in)$

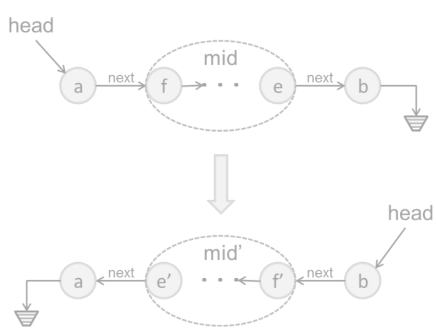
```
/* Average of x and y without using x+y (avoid overflow)*/
int avg(int x, int y){
  int t = expr({x/2, y/2, x%2, y%2, 2 }, {PLUS, DIV});
  assert t == (x+y)/2;
  return t;
}
```

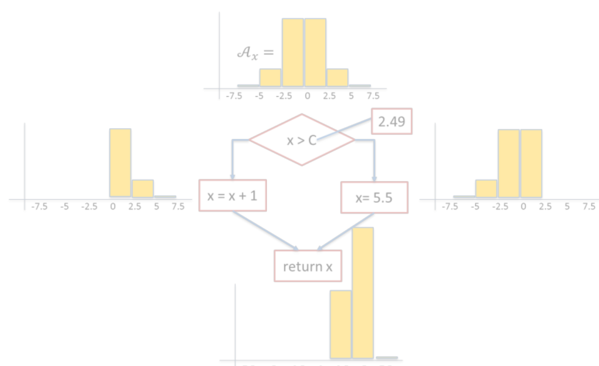





```
{
  s  = n.succ;
  p = n.pred;
  p.succ = s;
  s.pred = p;
}
```

Module I: Searching for Simple Programs



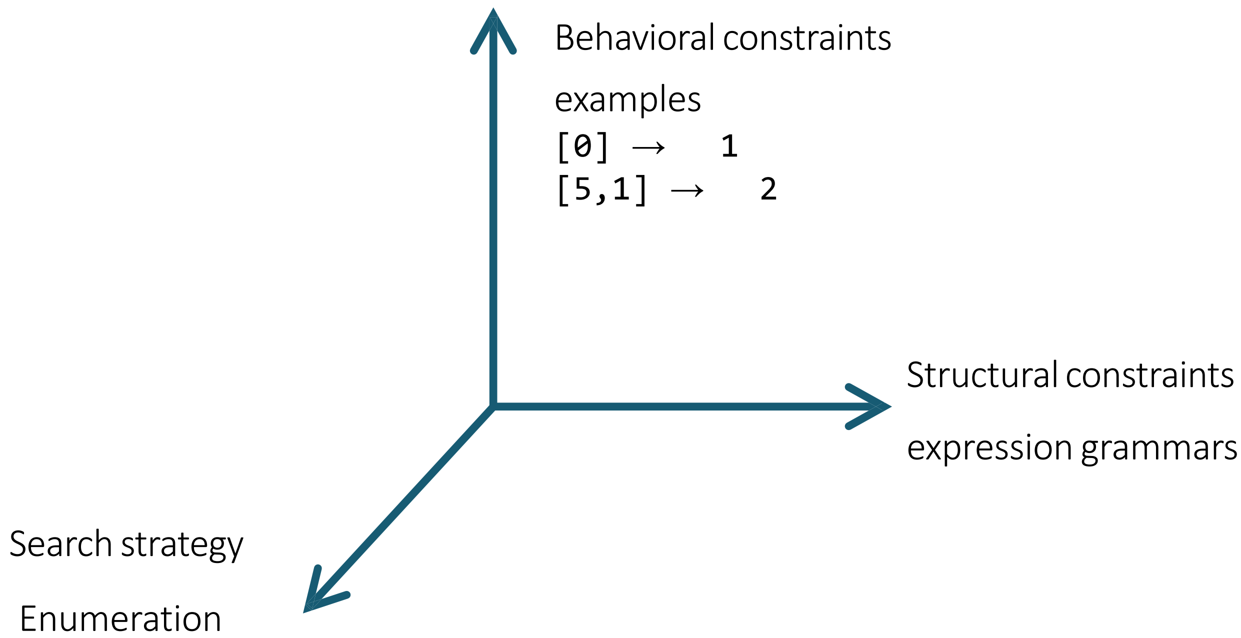


$\varphi(p)$

$Sk[c](in)$

Syntax-Guided Synthesis and Enumerative Search

Week 1-2


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

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

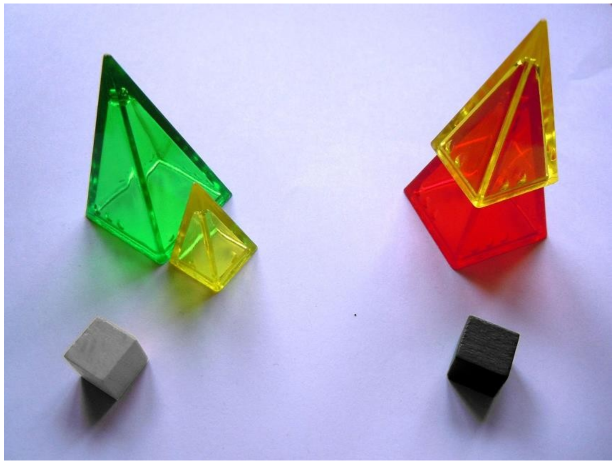
=

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

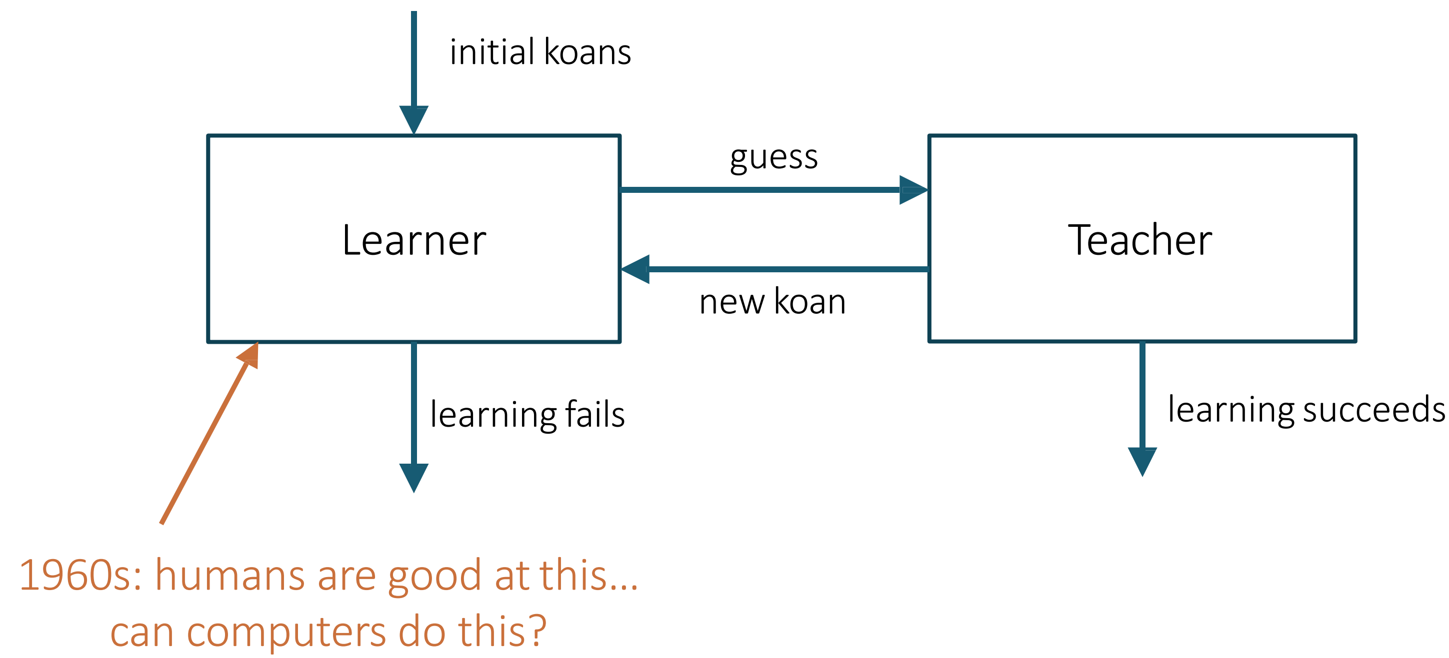
The teacher builds two koans (a positive and a negative)

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

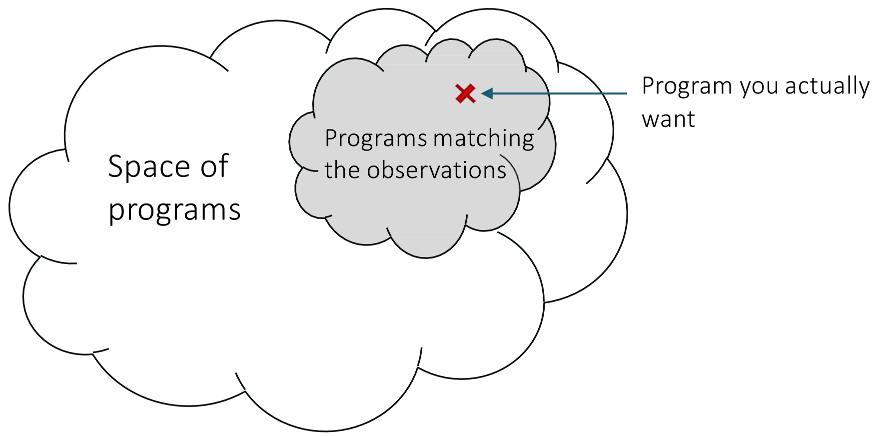
A student can try to guess the rule

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

The Zendo game



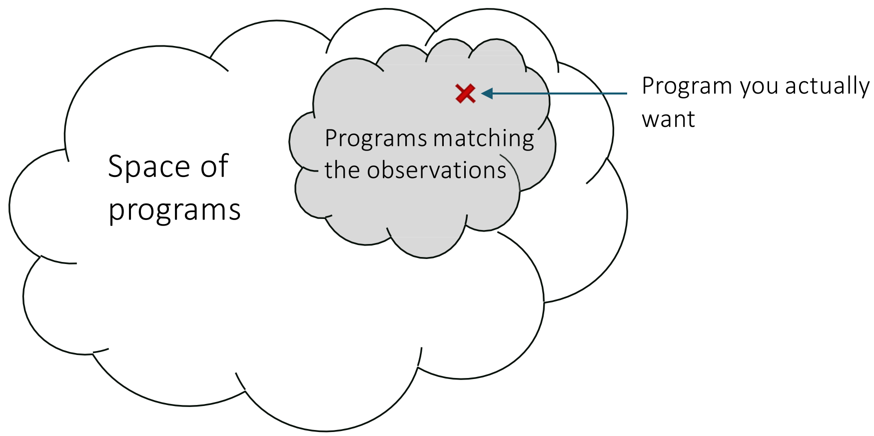
Key issues in inductive learning



(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)

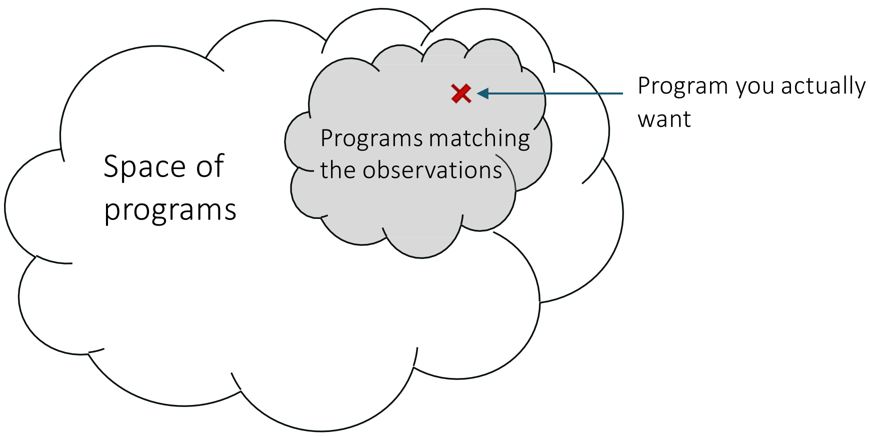
- Fix the space so that (1) is easy

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

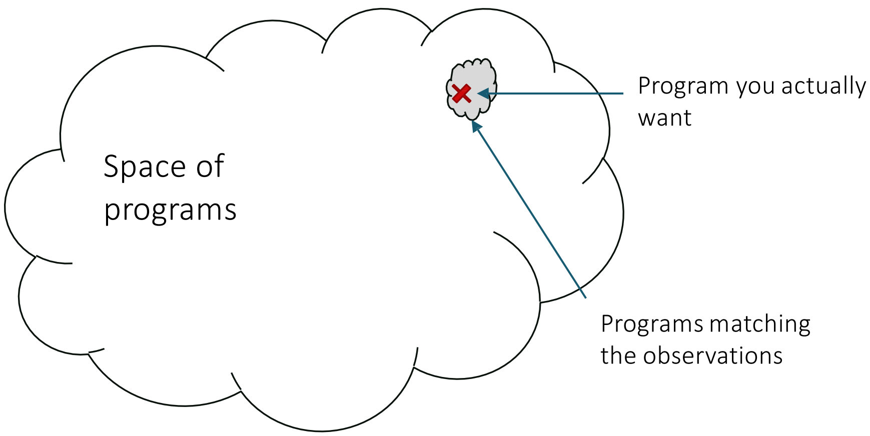


Modern emphasis

(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



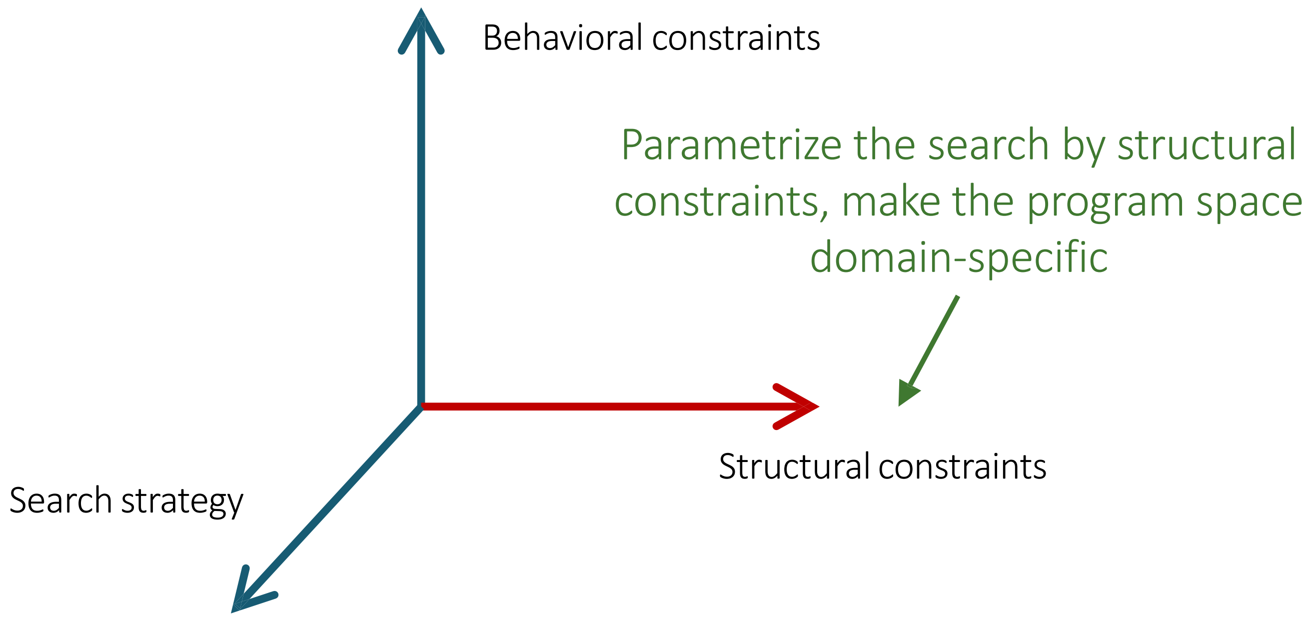
Modern emphasis

- 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


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

Please submit your Principles Of Programming Languages (He Zhu) of Spring 2021 Student Instructional Rating Survey by May 6!

Syntax-Guided Synthesis

Example

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

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

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

Context-free grammars (CFGs)

starting nonterminal

terminals

nonterminals

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

productions

CFGs as structural constraints

Space of programs
=
all ground, whole programs

```
(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]`

...

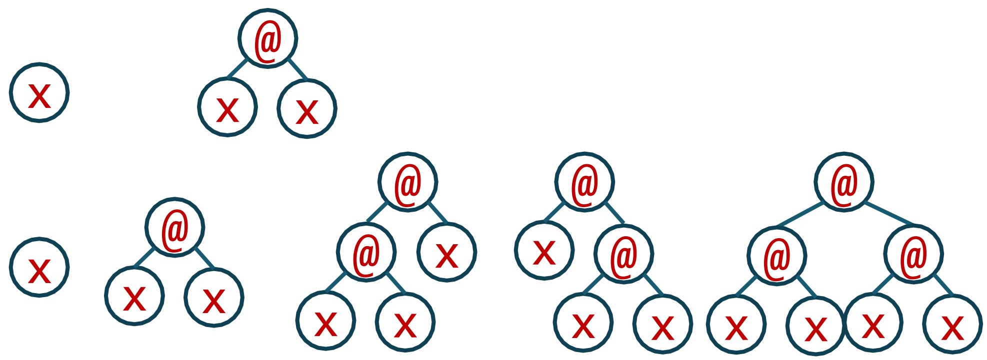
How big is the space?

`E ::= x | 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 | 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 |
      E @₁ E | ... | E @ₘ 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

$k = m = 3$

The SyGuS project

[Alur et al. 2013]

https://sygus.org/

Goal: Unify different syntax-guided approaches

Collection of synthesis benchmarks + yearly competition

- 7 competitions since 2013

Common input format + supporting tools

- parser, baseline synthesizers

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
```

CFG 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 problems

SyGuS problem = < theory, spec, grammar >

A first-order logic formula over the theory

can inductive synthesis handle these?

Examples:

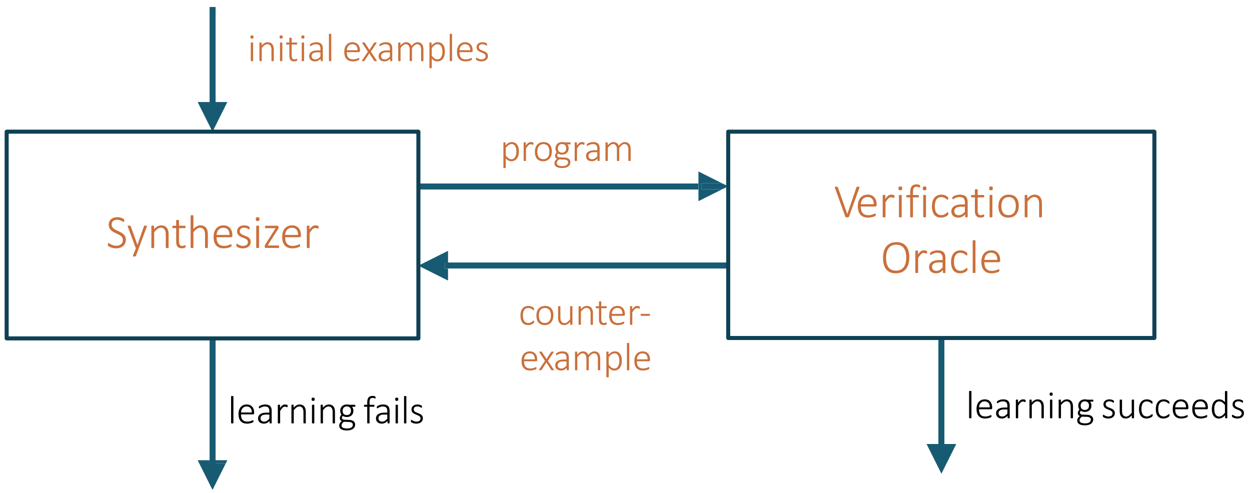
```
f(0, 1) = 1 ∧
f(1, 0) = 1 ∧
f(1, 1) = 1 ∧
f(2, 0) = 2
```

Formula with free variables:

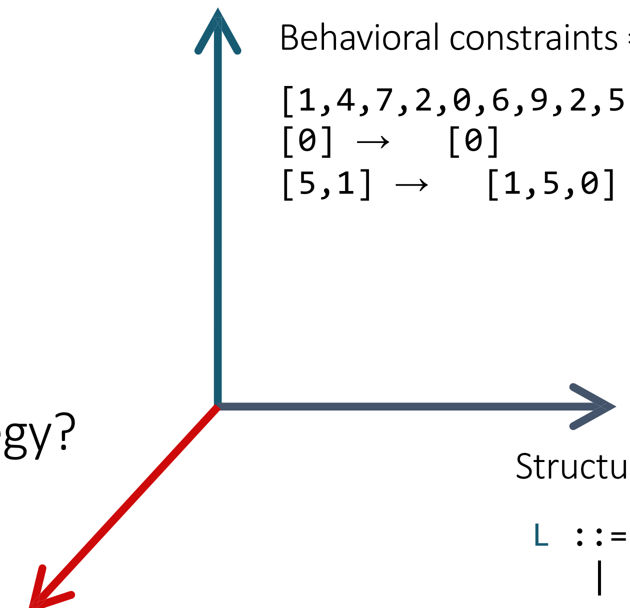
```
x ≤ f(x, y) ∧
y ≤ f(x, y) ∧
(f(x, y) = x ∨ f(x, y) = y)
```

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?

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] → [1,4]]`

Top-down enumeration

Start from the start non-terminal

Expand remaining non-terminals using productions

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

[[1,4,0,6]  →  [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]]`

Enumerative Search

Bottom-up

Top-down

Smaller to larger

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

How to make it scale

Prune

Discard useless subprograms

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

Prioritize

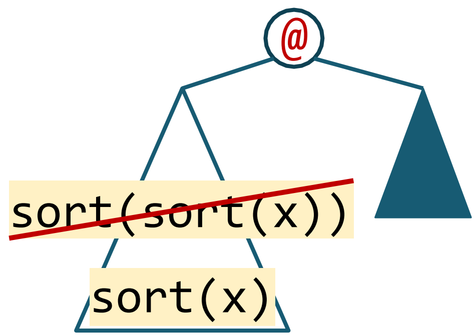
Explore more promising candidates first

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

dequeue this first

When can we discard a subprogram?

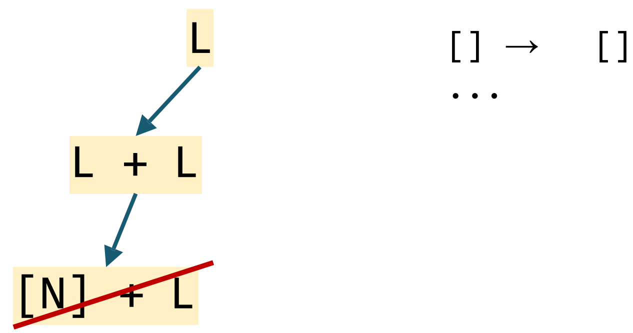
It's equivalent to something we have already explored



Equivalence reduction

(also: symmetry breaking)

No matter what we combine it with, it cannot satisfy the spec



Top-down propagation

Equivalent programs

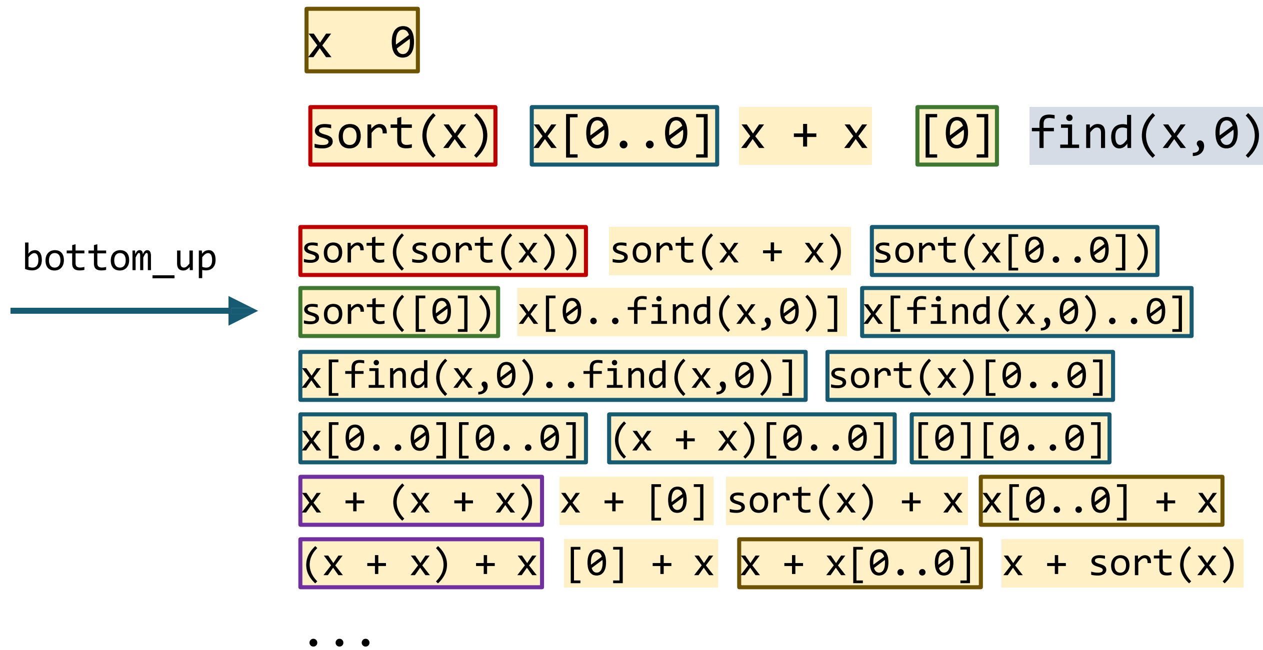
```
L ::= sort(L)    |
      L[N..N]    |
      L + L      |
      [N]        |
      x
N ::= find(L,N)  |
      0
```

bottom_up →

```
x   0

sort(x)  x[0..0]  x + x  [0]  find(x,0)

sort(sort(x))  sort(x + x)  sort(x[0..0])
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)
...
```

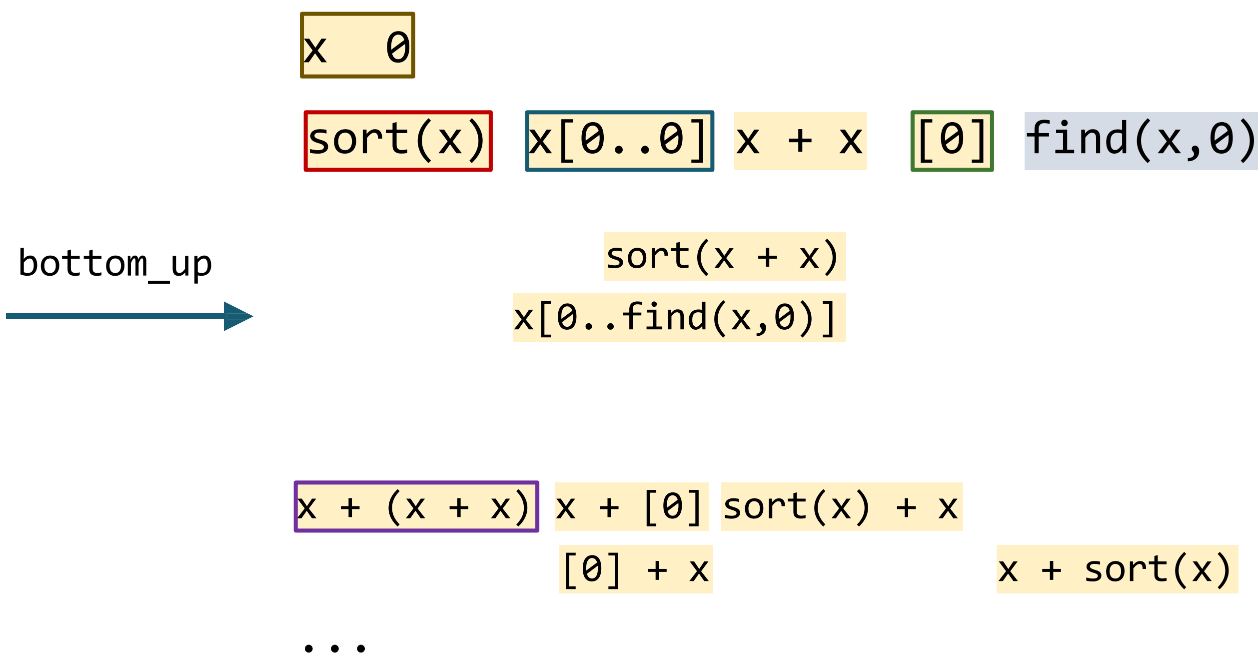
Equivalent programs

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



Equivalent programs

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



Observational equivalence

In PBE, all we care about is equivalence on the given inputs!

- easy to check efficiently
- even more programs are equivalent

`[[0] → [0]]`

`x` `0`

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

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

`x + (x + x)` `x + [0]` `sort(x) + x`
`[0] + x` `x + sort(x)`

Observational equivalence

In PBE, all we care about is equivalence on the given inputs!

- easy to check efficiently
- even more programs are equivalent

`[[0] → [0]]`

`x` `0`

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

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

`x + (x + x)` `x + [0]` `sort(x) + x`
`[0] + x` `x + sort(x)`

Observational equivalence

In PBE, all we care about is equivalence on the given inputs!

- easy to check efficiently
- even more programs are equivalent

`[[0] → [0]]`

`x` `0`

`x[0..0]` `x + x`

`x + (x + x)`

Observational equivalence

Proposed simultaneously in two papers:

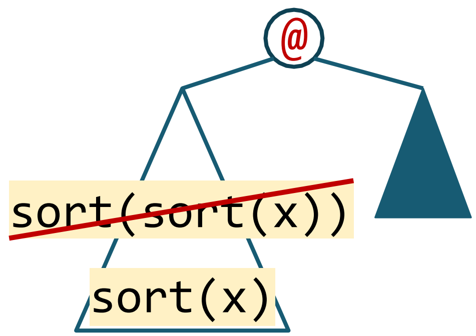
- Udupa, Raghavan, Deshmukh, Mador-Haim, Martin, Alur: TRANSIT: specifying protocols with concolic snippets. PLDI'13
- Albarghouthi, Gulwani, Kincaid: Recursive Program Synthesis. CAV'13

Variations used in most bottom-up PBE tools:

- ESolver (baseline SyGuS enumerative solver)
- Lens [Phothilimthana et al. ASLPOS'16]
- EUSolver [Alur et al. TACAS'17]

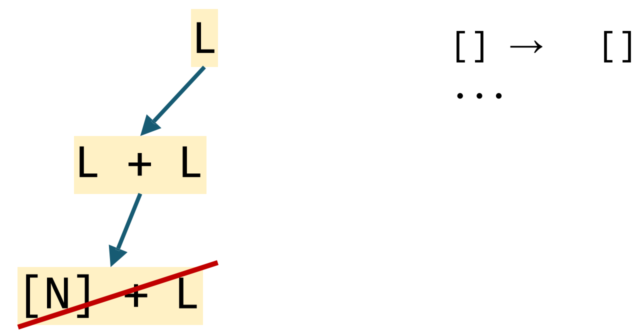
When can we discard a subprogram?

It's equivalent to something we have already explored



Equivalence reduction

No matter what we combine it with, it cannot fit the spec



Top-down propagation

Top-down search: reminder

generates a lot of non-ground terms
only discards ground terms

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))]` … … … …

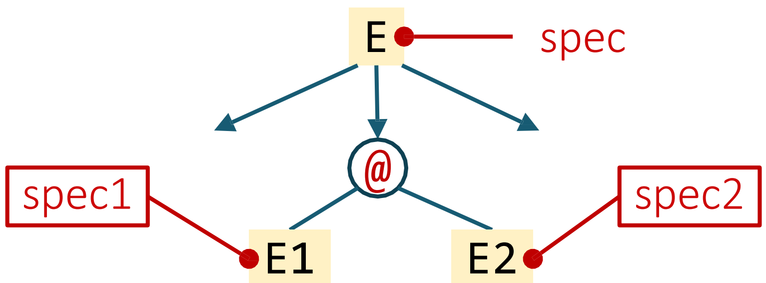
iter 9:

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

`[[1,4,0,6] → [1,4]]`

Top-down propagation

Idea: once we pick the production, infer specs for subprograms

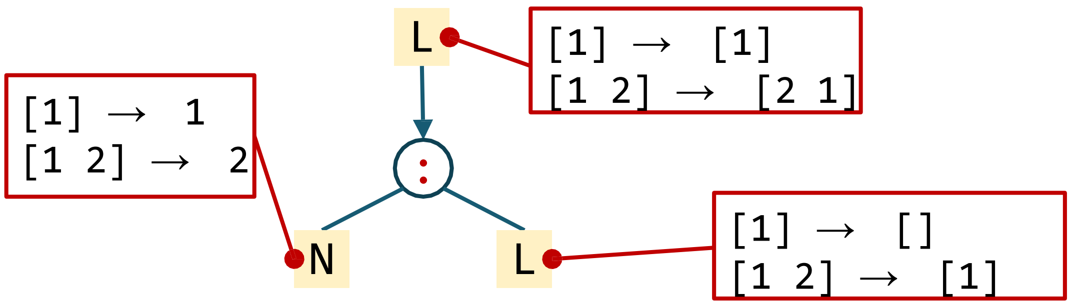


If `spec1 = ⊥`, discard `E1 @ E2` altogether!

For now: spec = examples

When is TDP possible?

Depends on @!

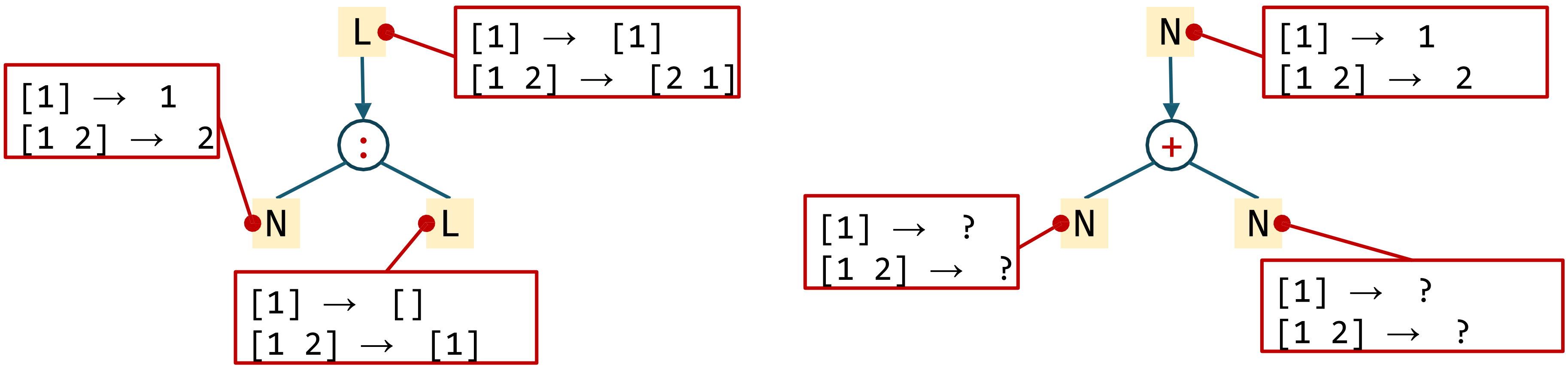


Works when the function is injective!

Q: when would we infer ⊥? A: If at least one of the outputs is [ ]!

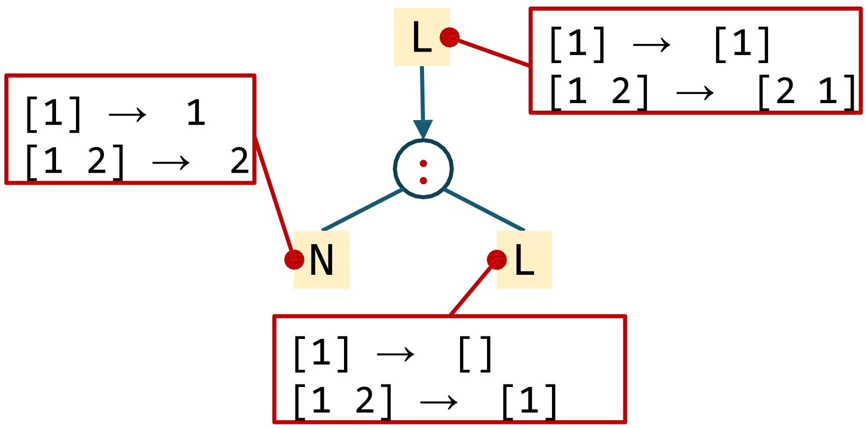
When is TDP possible?

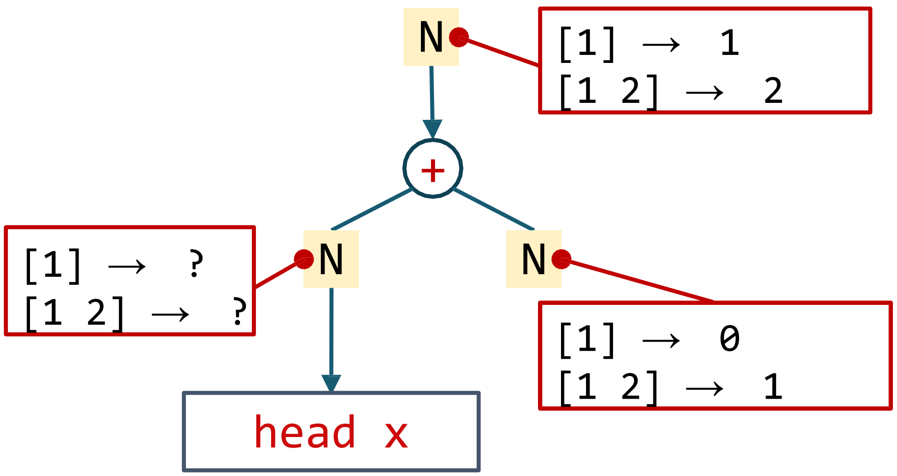
Depends on @!



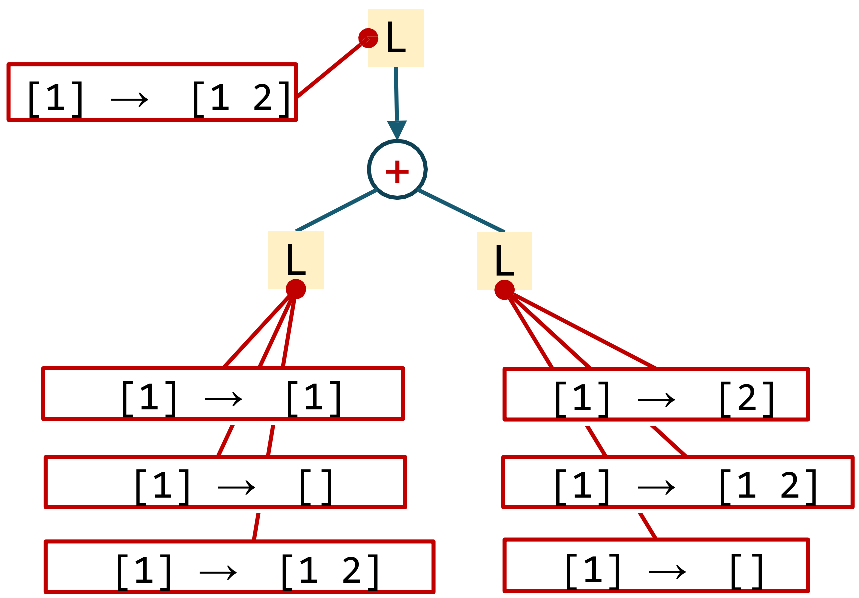
When is TDP possible?

Depends on @!





Something in between?



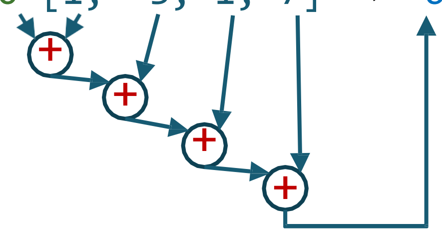
Works when the function is "sufficiently injective"

- output examples have a small pre-image

λ²: TDP for list combinators

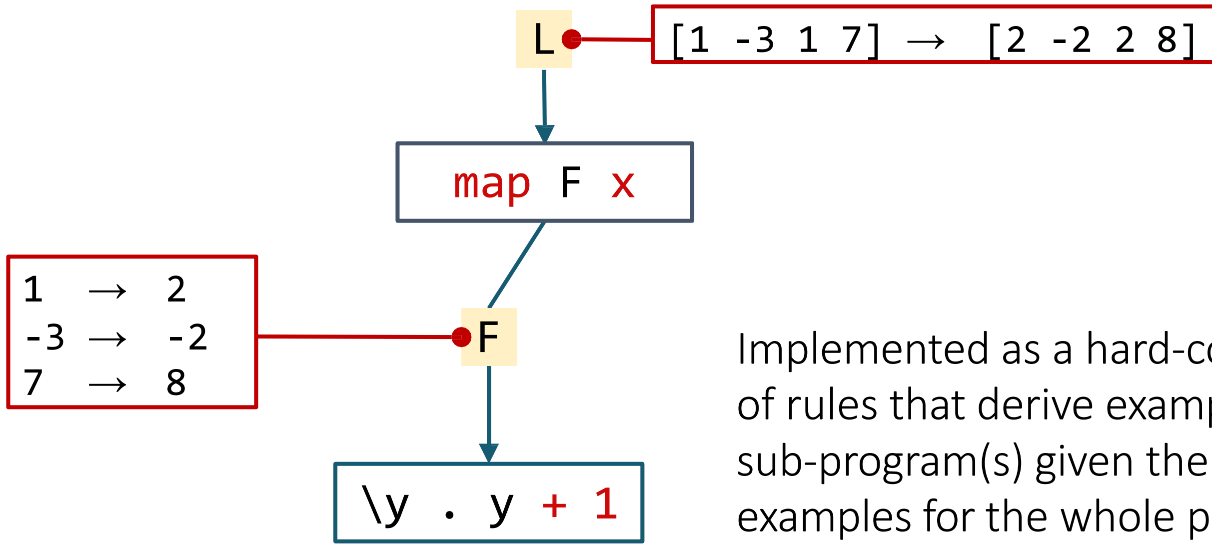
[Feser, Chaudhuri, Dillig '15]

```
map f x              map (\y . y + 1) [1, -3, 1, 7] → [2, -2, 2, 8]

filter f x           filter (\y . y > 0) [1, -3, 1, 7] → [1, 1, 7]

fold f acc x         fold (\y z . y + z) 0 [1, -3, 1, 7] → 6
```



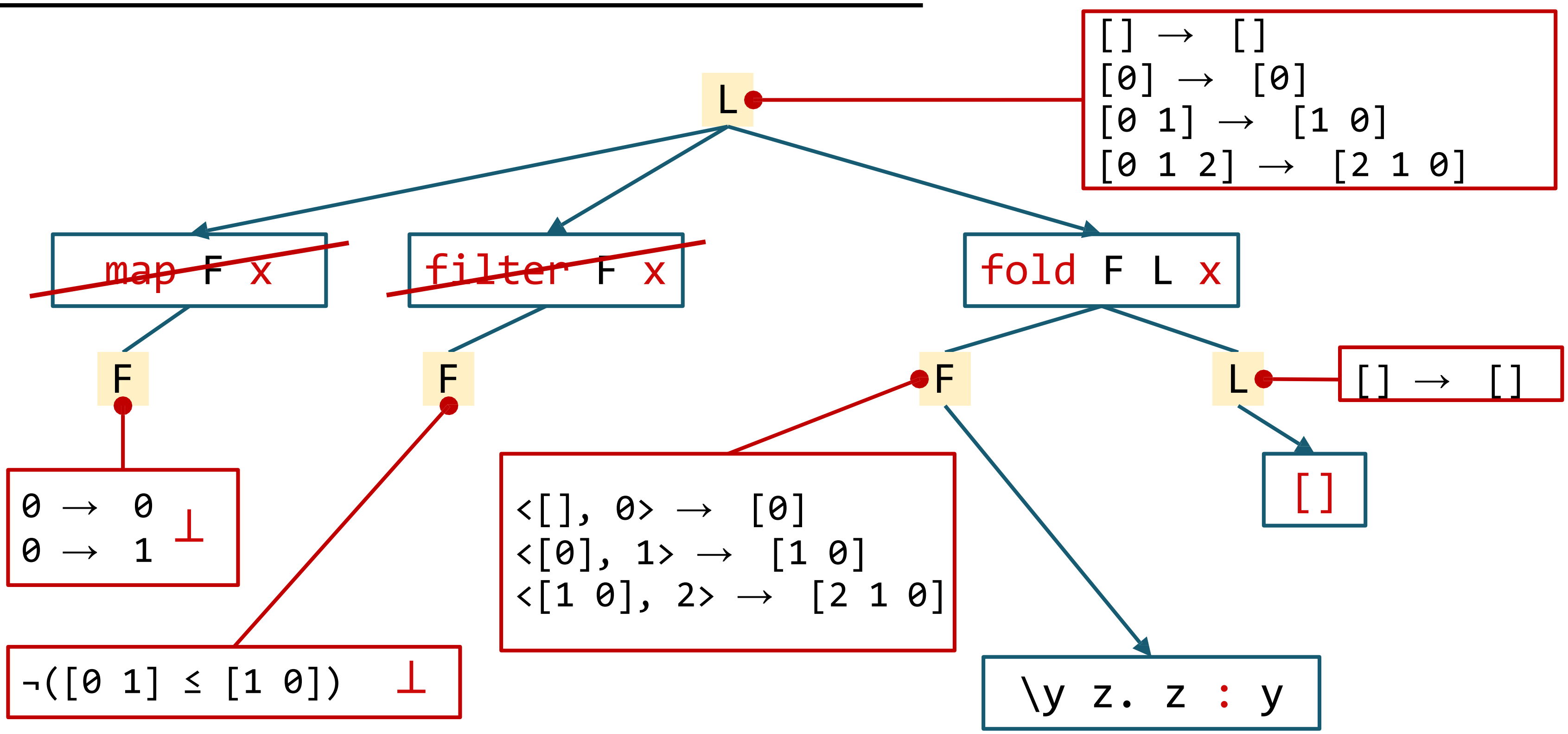
```
                     fold (\y z . y + z) 0 [] → 0
```

λ²: TDP for list combinators



Implemented as a hard-coded set of rules that derive examples for sub-program(s) given the examples for the whole program and the combinator

λ²: TDP for list combinators



Condition abduction

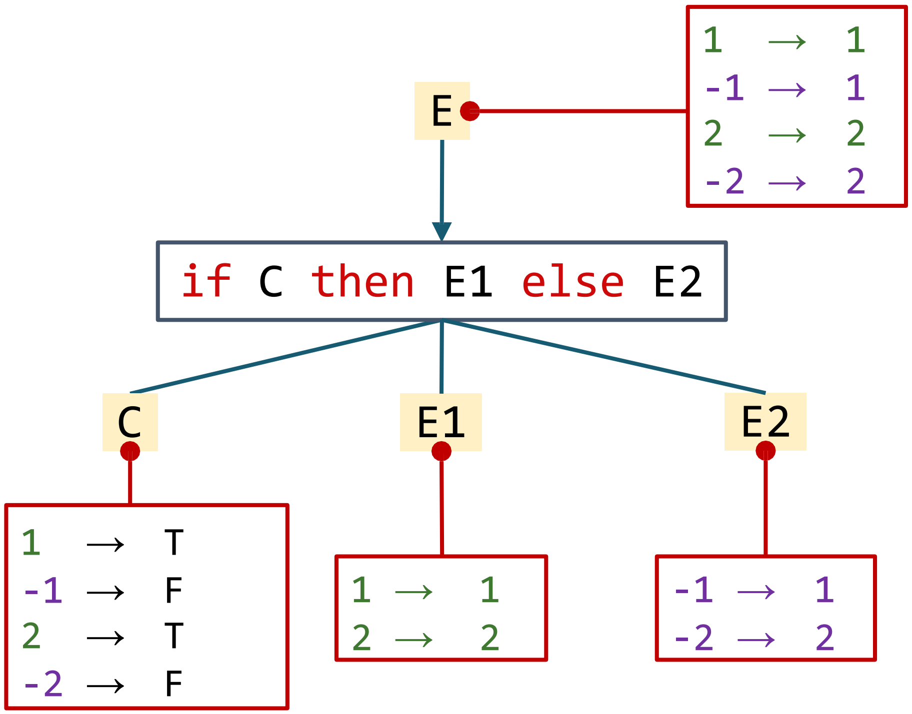
Smart way to synthesize conditionals

Used in many tools (under different names):

- FlashFill [Gulwani ‘11]
- Escher [Albarghouthi et al. ‘13]
- Leon [Kneuss et al. ‘13]
- Synquid [Polikarpova et al. ‘13]
- EUSolver [Alur et al. ‘17]

In fact, an instance of TDP!

Condition abduction



Q: How does EUSolver decide how to split the inputs?

Q: How does EUSolver generate C?

How to make it scale

Prune

Discard useless subprograms

m * N² m * (N - 1)²

Prioritize

Explore more promising candidates first

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

← dequeue this first

End of the course! Thank you!

Please submit your Principles Of Programming Languages (He Zhu) of Spring 2021 Student Instructional Rating Survey by May 6!