Lecture 3
Search Space Pruning

Nadia Polikarpova

Logistics

Reviews

- due tomorrow

Project

- teams due Friday
- who hasn't found a team yet?

Today

Pruning techniques for enumerative search

- Equivalence reduction
- Top-down specification propagation

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

=

Explicit / Exhaustive Search

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

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

top-down

```
L

x   sort(L)   L[N..N]   L + L   [N]

sort(x)   sort(sort(L))   sort([N])
sort(L[N..N])   sort(L + L)
x[N..N]   (sort L)[N..N]   ...
```

How to make it scale

Prune

Discard useless subprograms

m * N²   m * (N - 1)²

Prioritize

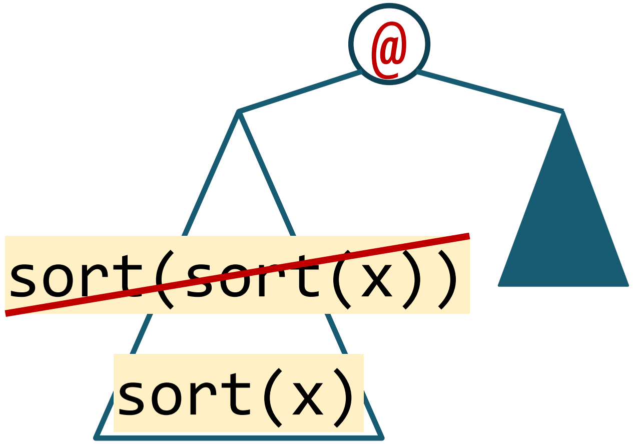
Explore more promising candidates first

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

dequeue this first

When can we discard a subprogram?

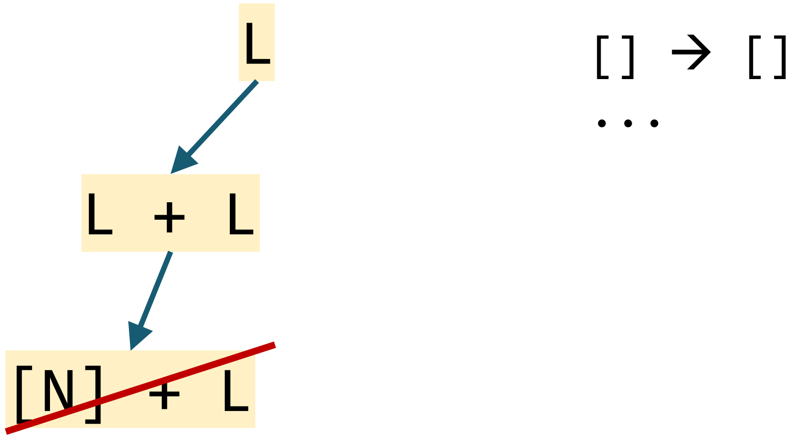
It's equivalent to something we have already explored



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
```

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
```

bottom_up →

```
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)

...
```

Bottom-up + equivalence reduction

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

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

How do we implement **equiv**?

- In general undecidable
- For SyGuS problems: expensive
- Doing expensive checks on every candidate defeats the purpose of pruning the space!

Observational equivalence

```
bottom-up (<T, N, R, S>, [i → o])
{ ... }

equiv(p, p') {
  return p([i]) = p'([i])
}
```

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

```
bottom-up (<T, N, R, S>, [i → o])
{ ... }

equiv(p, p') {
  return p([i]) = p'([i])
}
```

```
[[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

```
bottom-up (<T, N, R, S>, [i → o])
{ ... }

equiv(p, p') {
  return p([i]) = p'([i])
}
```

`[[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]
- **Probe** [Barke et al. OOPSLA’20]

User-specifies equations

[Smith, Albarghouthi: VMCAI'19]

Equations

```
sort(sort(l)) = sort(l)
(l1 + l2) + l3 = l1 + (l2 + l3)
n = n + 0
n + m = m + n
```

derived automatically →

Term-rewriting system (TRS)

1. `sort(sort(l))` → `sort(l)`
2. `(l1 + l2) + l3` → `l1 + (l2 + l3)`
3. `n + 0` → `n`
4. `n + m` $\rightarrow_{(n > m)}$ `m + n`

`x` `0`

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

~~`sort(sort(x))`~~ rule 1 applies, not in *normal form*

Built-in equivalences

For a predefined set of operations, equivalence reduction can be hard-coded in the tool or built into the grammar

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

→

```
L  ::= L1  |  L1 + L
L1 ::= sort(L)   |
       L[N..N]   |
       [N]       |
       x
N  ::= find(L,N) |
      0
```

Built-in equivalences

Used by:

- **λ^2** [Feser et al.'15]
- **Leon** [Kneuss et al.'13]

Leon's implementation using *attribute grammars* described in:

- Koukoutos, Kneuss, Kuncak: An Update on Deductive Synthesis and Repair in the Leon tool [SYNT'16]

Equivalence reduction: comparison

Observational

- Very general, no user input required
- Finds more equivalences
- Can be costly (with many examples, large outputs)
- If new examples are added, has to restart the search

User-specified

- Fast
- Requires equations

Built-in

- Even faster
- Restricted to built-in operators
- Only certain symmetries can be eliminated by modifying the grammar

Q1: Can any of them apply to top-down?

Q2: Can any of them apply beyond PBE?

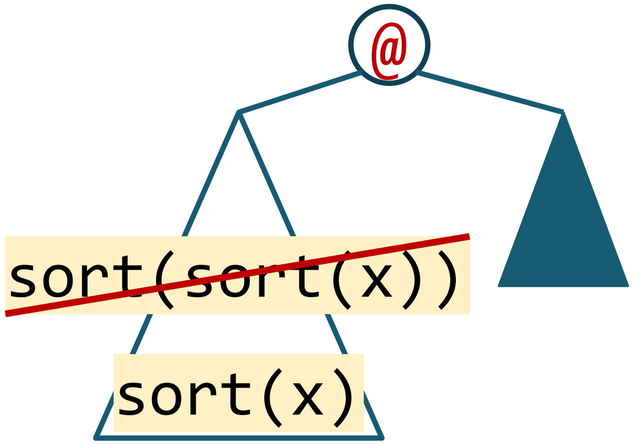
Today

Top-down Propagation

EUSolver discussion

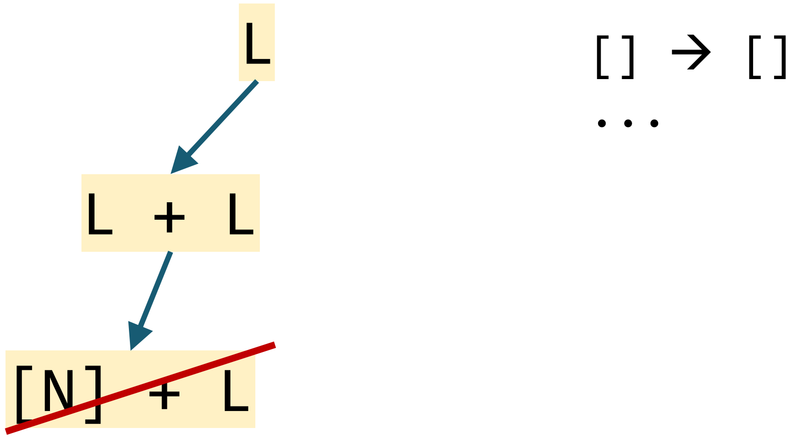
When can we discard a subprogram?

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))]` … … … …

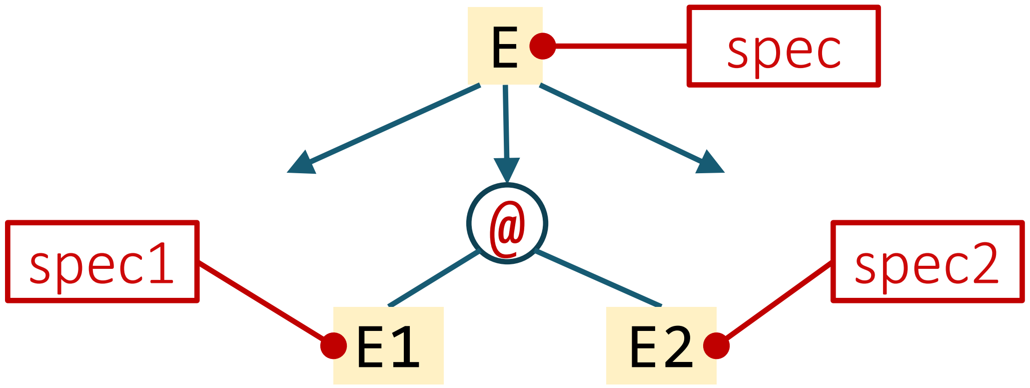
iter 9:

```
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

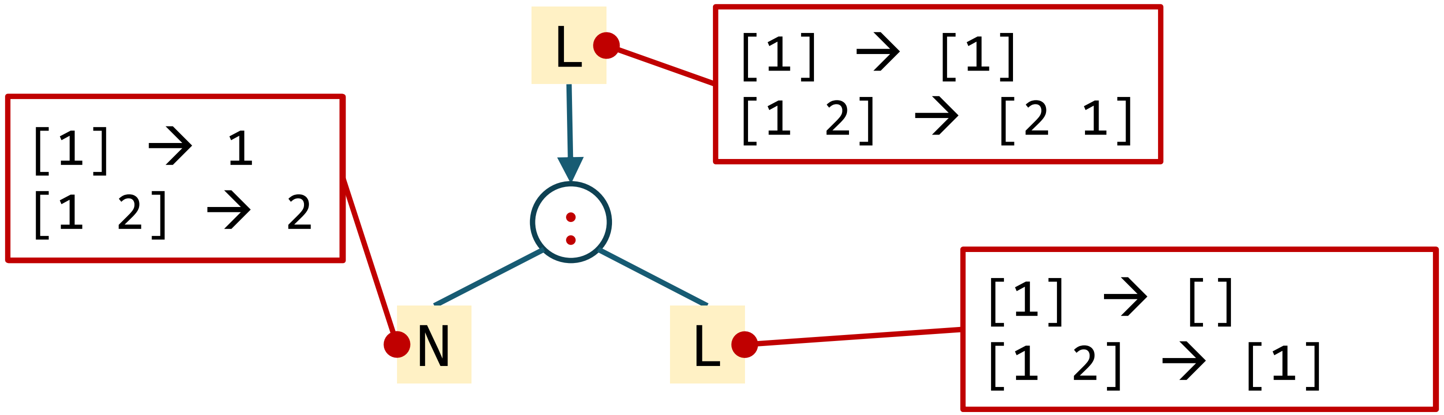


If `spec1 = ⊥`, discard `E1 @ E2` altogether!

For now: spec = examples

When is TDP possible?

Depends on @!

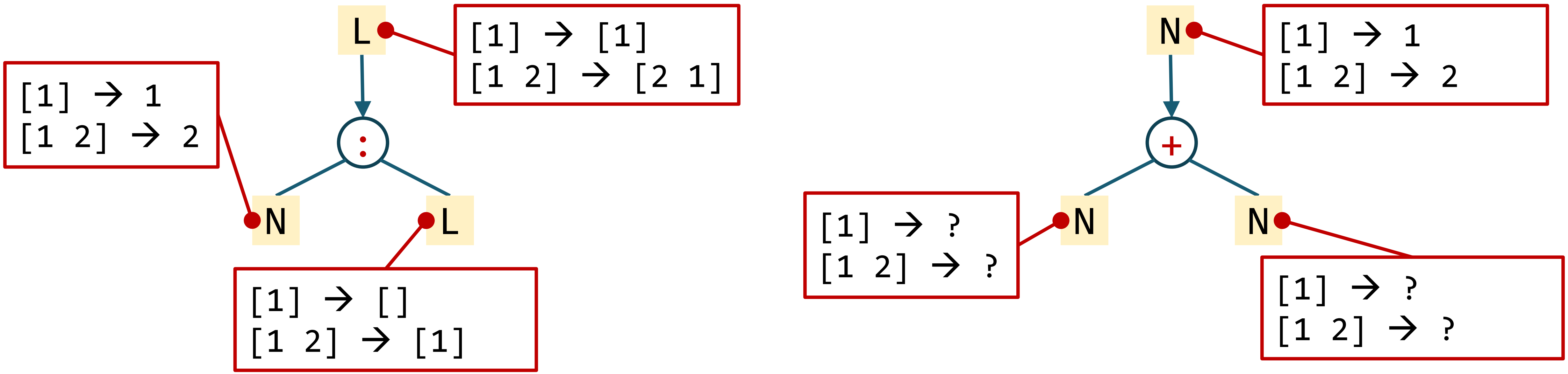


Works when the function is injective!

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

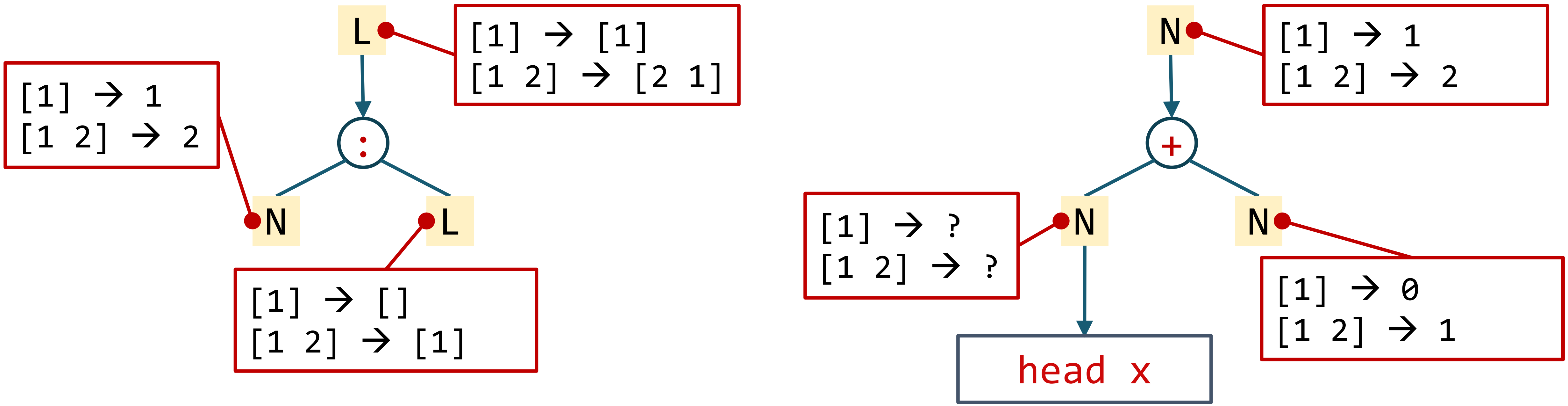
When is TDP possible?

Depends on @!

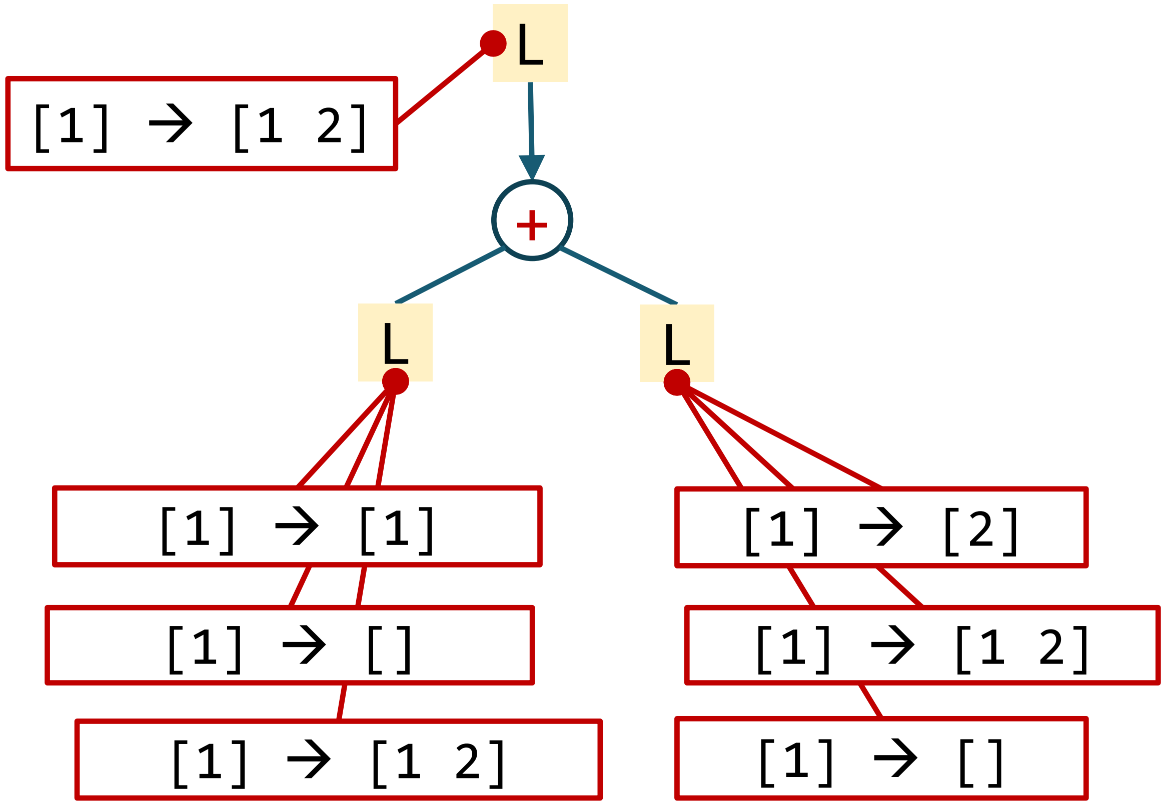


When is TDP possible?

Depends on @!



Something in between?



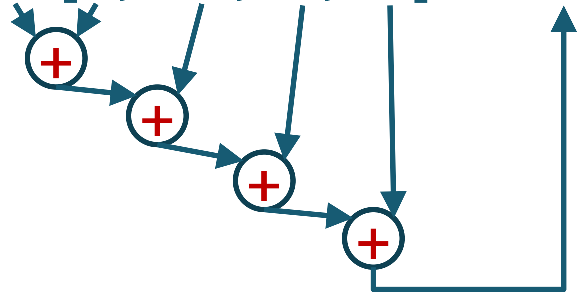
Works when the function is "sufficiently injective"

- output examples have a small pre-image

λ²: TDP for list combinators

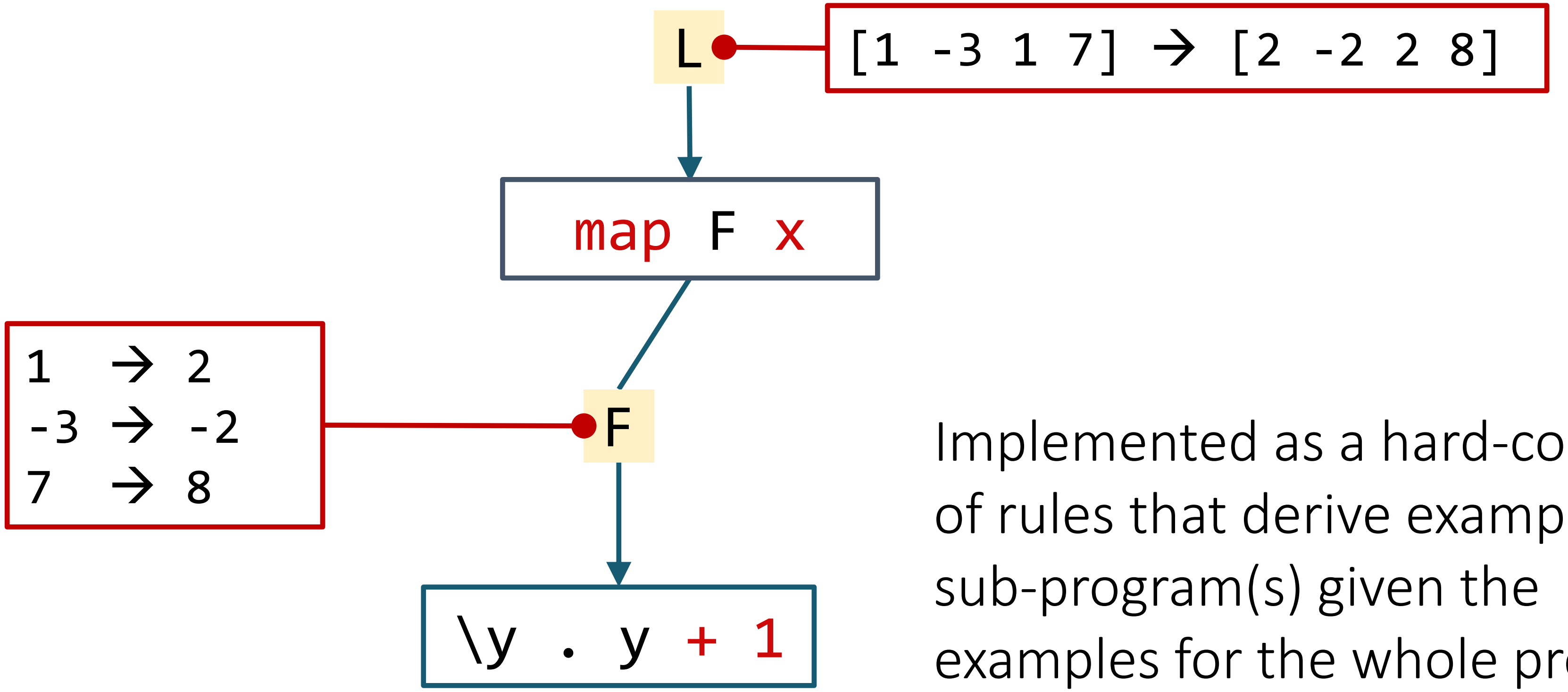
[Feser, Chaudhuri, Dillig '15]

```
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
```



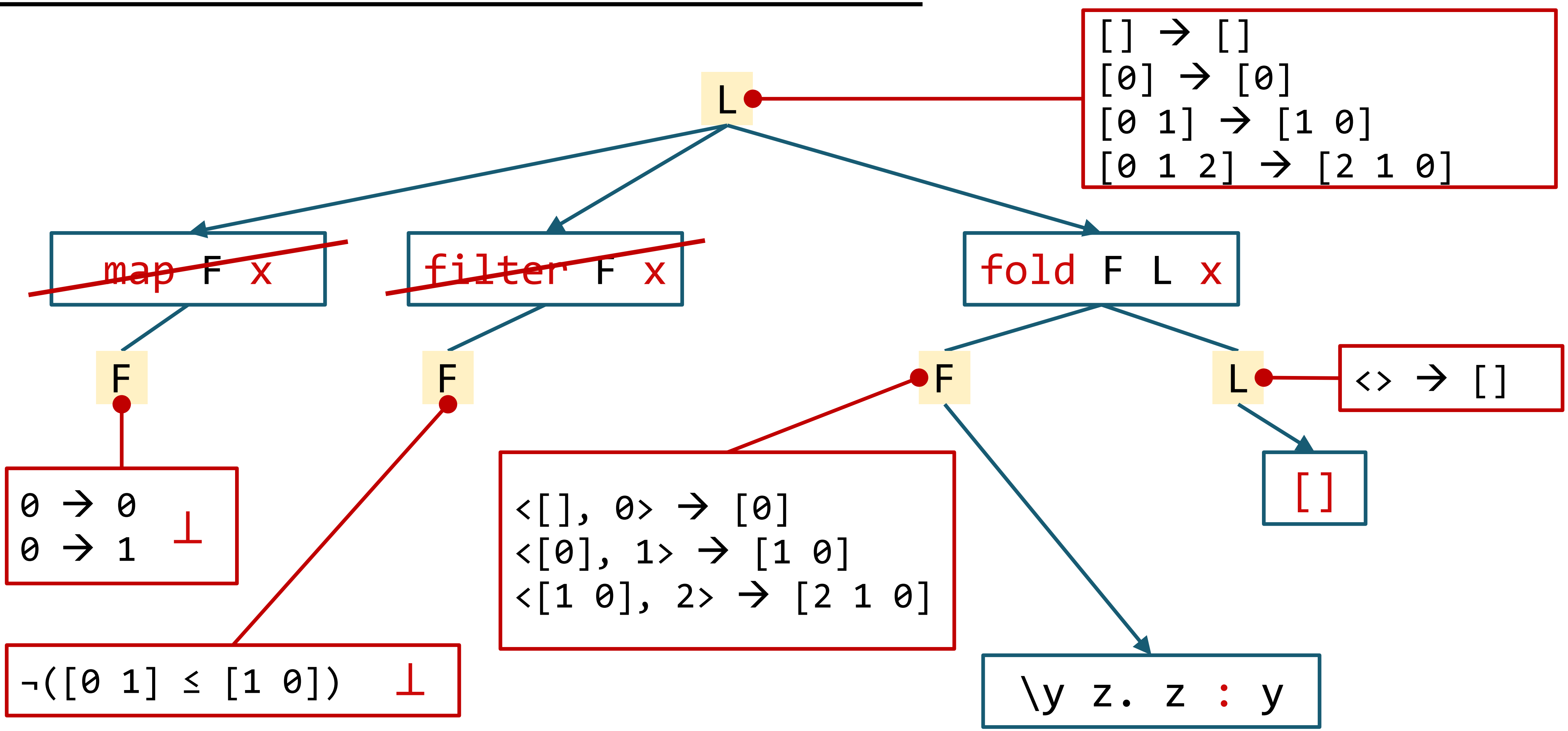
```
                    fold (\y z . y + z) 0 [] → 0
```

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

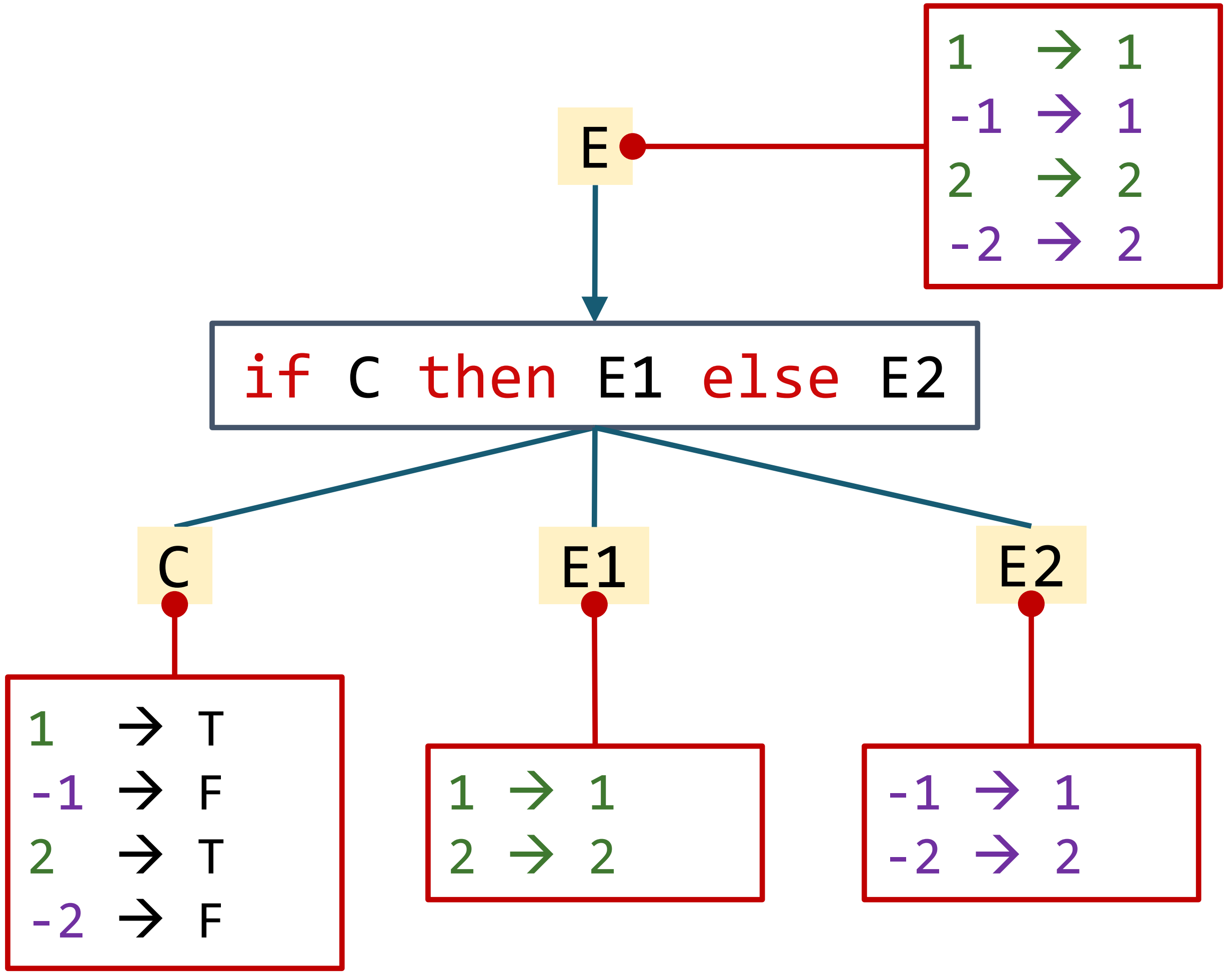
Smart way to synthesize conditionals

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



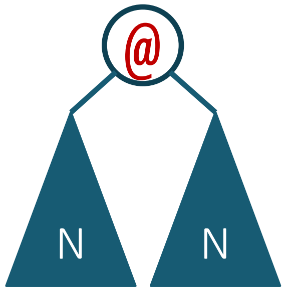
Q: How does EUSolver decide how to split the inputs?

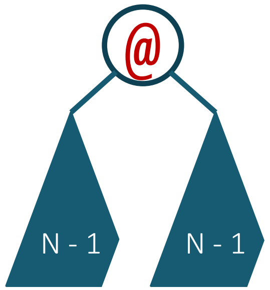
Q: How does EUSolver generate C?

How to make it scale

Prune

Discard useless subprograms





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

Prioritize

Explore more promising candidates first

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

dequeue this first

Feedback on reviews

More discussion of the technique/eval and less of the writing:

- good: “A major weakness of the this work is its restrictive scope: it only applies to synthesis of conditional expressions.”
- bad: “Graphs are easy to read.”

EUSolver

Q1: What does EUSolver use as behavioral constraints? Structural constraint? Search strategy?

- First-order formula
- Conditional expression grammar
- Bottom-up enumerative + pruning

Why do they need the specification to be pointwise?

- Example of a non-pointwise spec?
- How would it break the enumerative solver?

EUSolver

Q2: What are pruning/decomposition techniques EUSolver uses to speed up the search?

- Condition abduction + special form of equivalence reduction

Why does EUSolver keep generating additional terms when all inputs are covered?

How is the EUSolver equivalence reduction differ from observational equivalence we saw in class?

- How do they overcome the problem that it's not robust to adding new points?

Can we discard a term that covers a subset of the points covered by another term?

EUSolver

Q3: What would be a naive alternative to decision tree learning for synthesizing branch conditions?

- Learn atomic predicates that precisely classify points
  - why is this worse?
  - is it as bad as ESolver?
- Next best thing is decision tree learning w/o heuristics
  - why is this worse?

EUSolver: strengths

Divide-and-conquer (aka condition abduction)

- scales better on conditional expressions
- but: they didn't invent it

Neat application of decision tree learning

- leverages the structure of Boolean expressions

EUSover: weaknesses

Only applies to conditional expressions

Does not always generate the smallest expression

- but eventually it's always as good as enumerative solver

Only works for pointwise specifications

- but so do ALL CEGIS-based approaches