Lecture 3
Scaling Enumerative Search

Nadia Polikarpova

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

How to make it scale

Prune

Discard useless subprograms

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

Prioritize

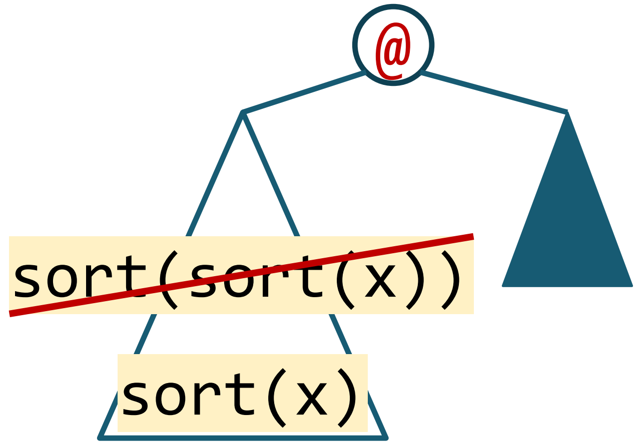
Explore more promising candidates first

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

dequeue this first

When can we discard a subprogram?

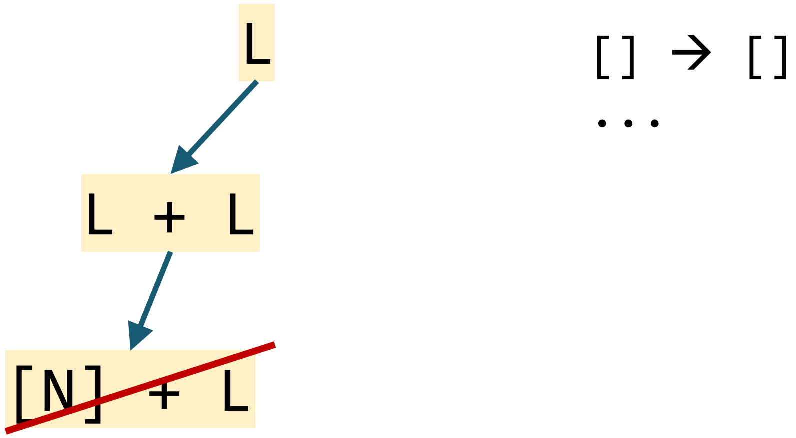
It's equivalent to something we have already explored



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]) {
  P := [t | t in T && t is nullary]
  while (true)
    P += grow(P);
    P := reduce(P);
    forall (p in P)
      if (whole(p) && p([i]) = [o])
        return p;
}
reduce(P) {
  P' := []
  forall (p in P)
    r := exists p' in P': equiv(p, p');
    if !r
      P' += p;
  return 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])
}
```

```
[[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)
```

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

- easy to check efficiently
- even more programs are equivalent

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)
```

Used in almost all PBE tools:
ESolver [Udupa et al. '13]
Escher [Albarghouthi et al. '13]
Lens [Phothilimthana et al. '16]
EUSolver [Alur et al. '17]
...

User-specifies equivalences

[Smith, Albarghouthi: unpublished]

Equivalences

```
sort(sort(l)) = sort(l)
(l + l) + l = l + (l + l)
n = n + 0
n + m = m + n
```

derived automatically

Term-rewriting system (TRS)

1. `sort(sort(l))` $\rightarrow$ `sort(l)`
2. `(l + l) + l` $\rightarrow$ `l + (l + l)`
3. `n + 0` $\rightarrow$ `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 *normal*

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
```

Used by **Leon** [Kneuss et al.'13], λ^2 [Feser et al.'15], ...

Equivalence reduction: comparison

Observational

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

User-specified

- Fast: no need to call `reduce`

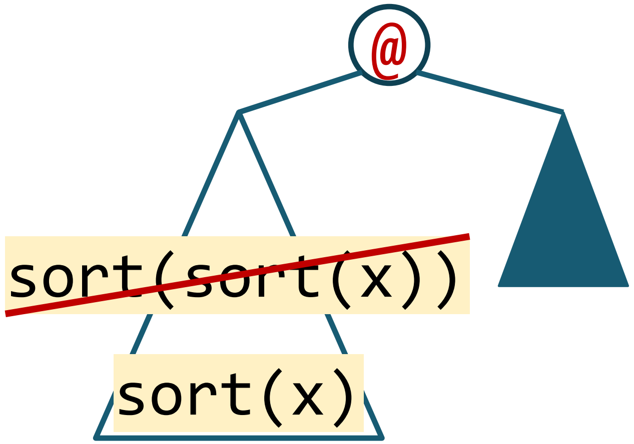
Built-in

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

Can any of them apply to top-down?

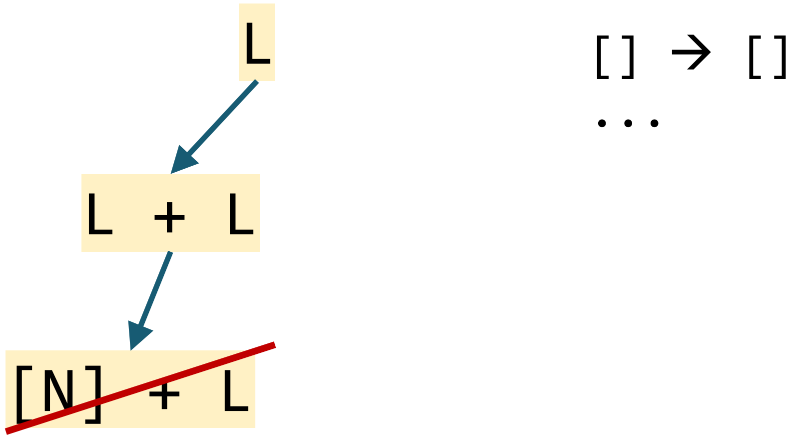
When can we discard a subprogram?

It’s equivalent to something we have already explored



Equivalence reduction

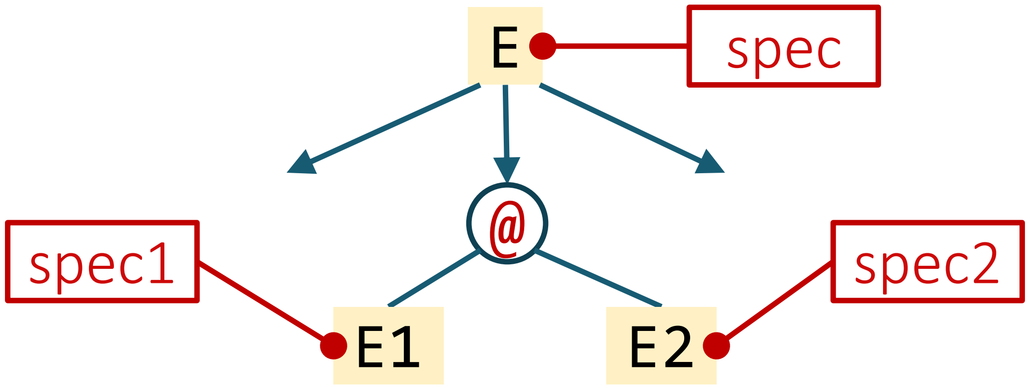
No matter what we combine it with, it cannot fit the spec



Top-down propagation

Top-down propagation

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

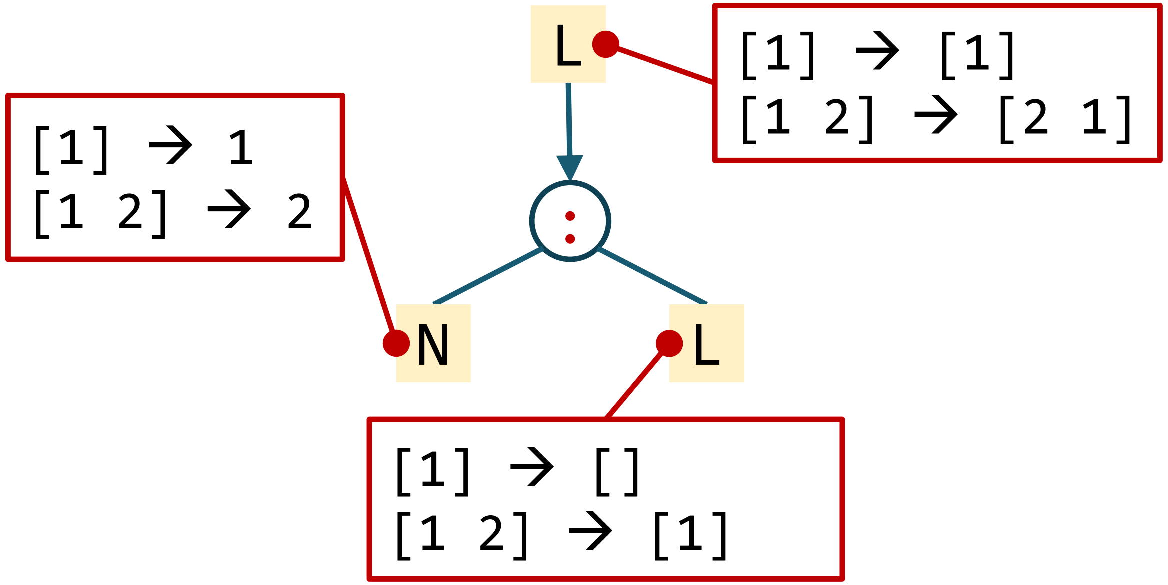


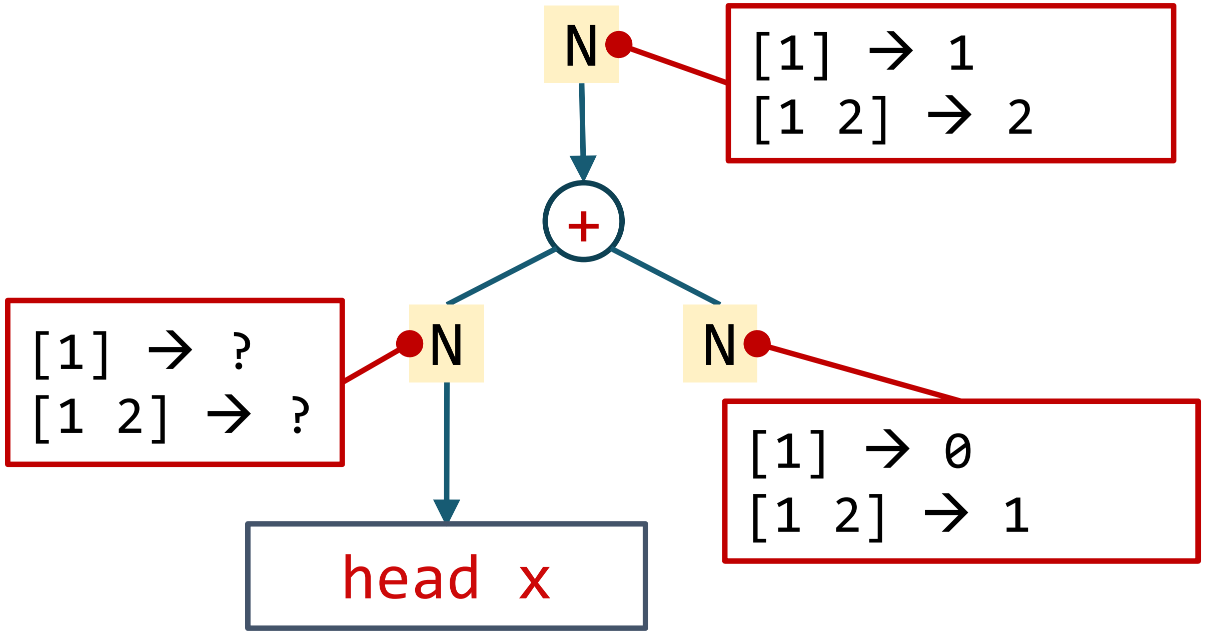
If `spec1 = ⊥`, discard `E1 @ E2` altogether!

For now: spec = examples

When is TDP possible?

Depends on @!





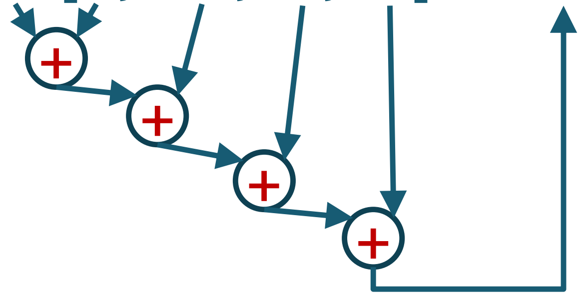
Q: when would we infer ⊥?

Great for injective functions

λ²: TDP for list combinators

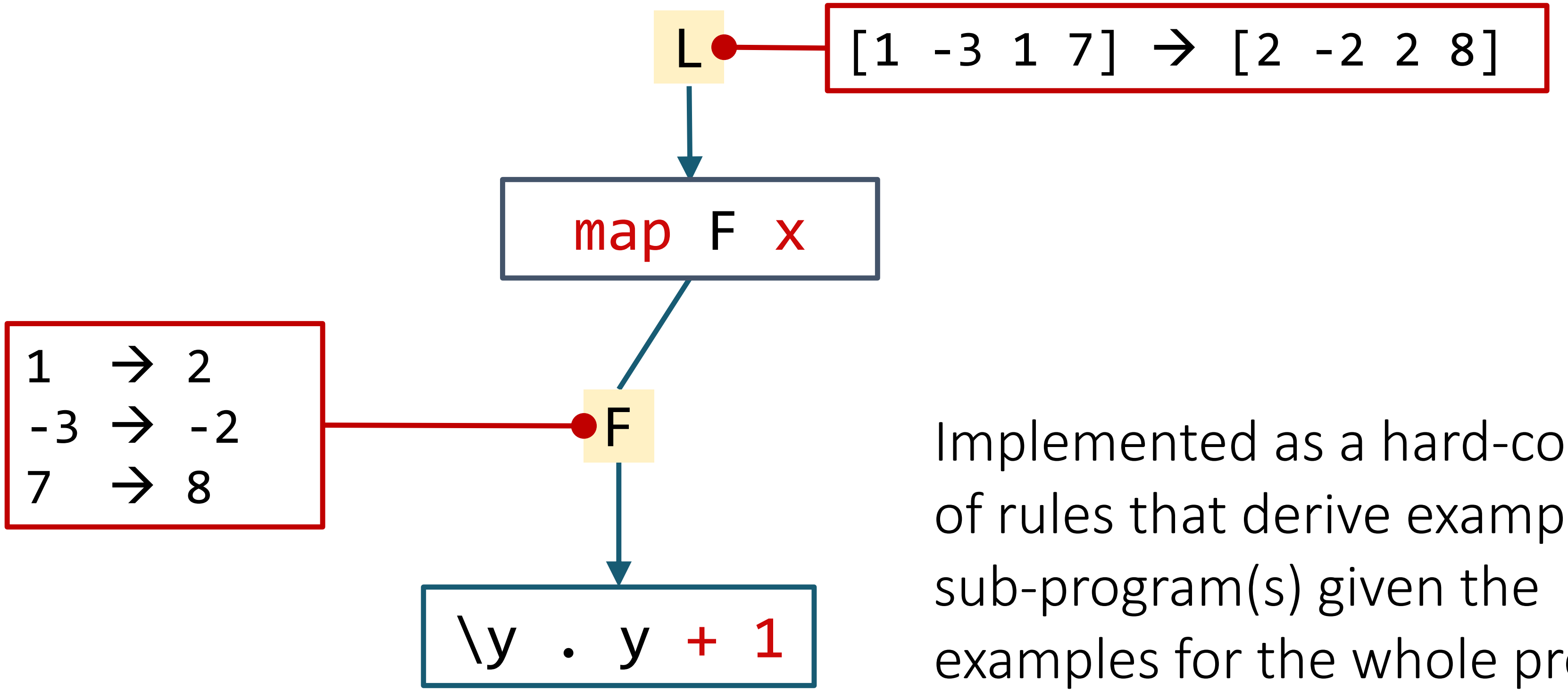
[Feser, Chaudhuri, Dillig '15]

```
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
```



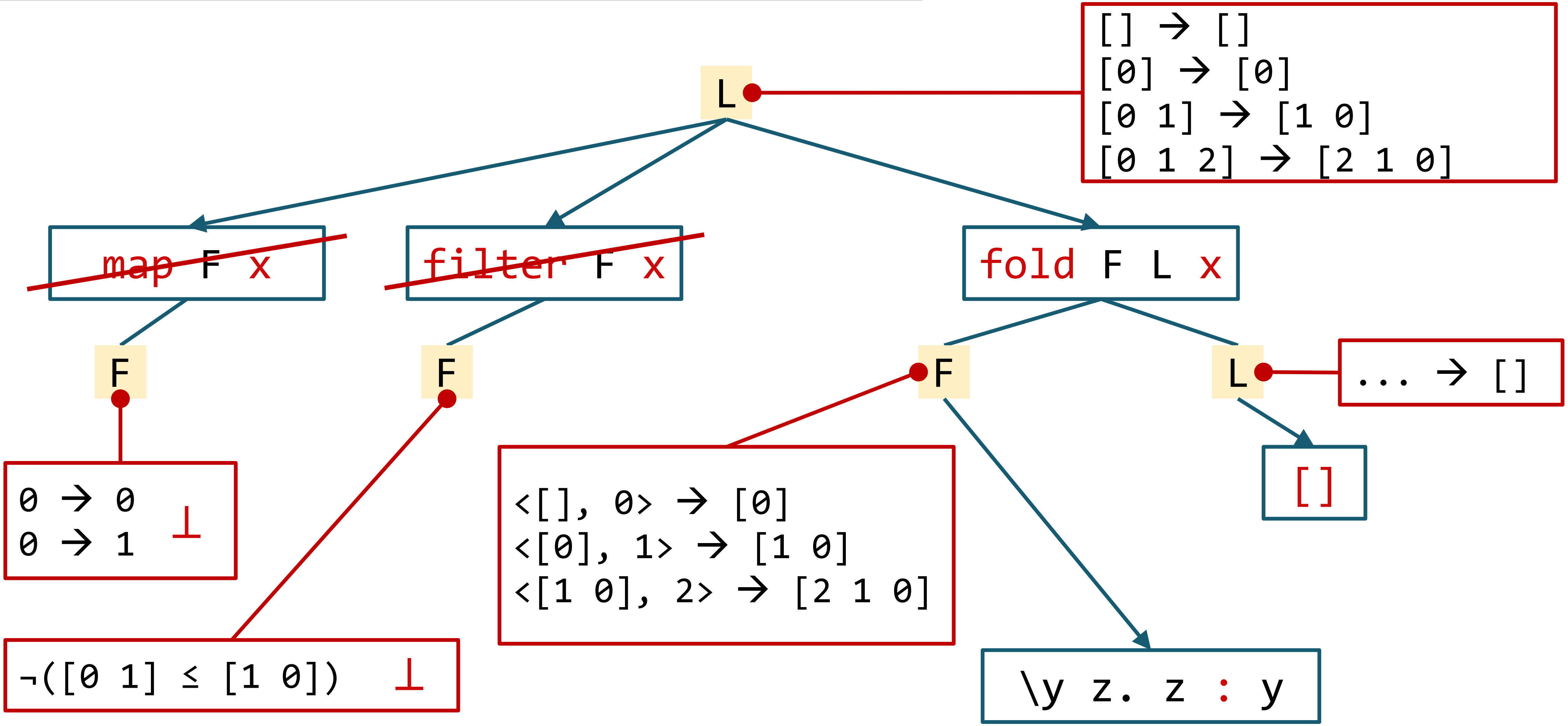
```
                     fold (\y z . y + z) 0 [] → 0
```

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

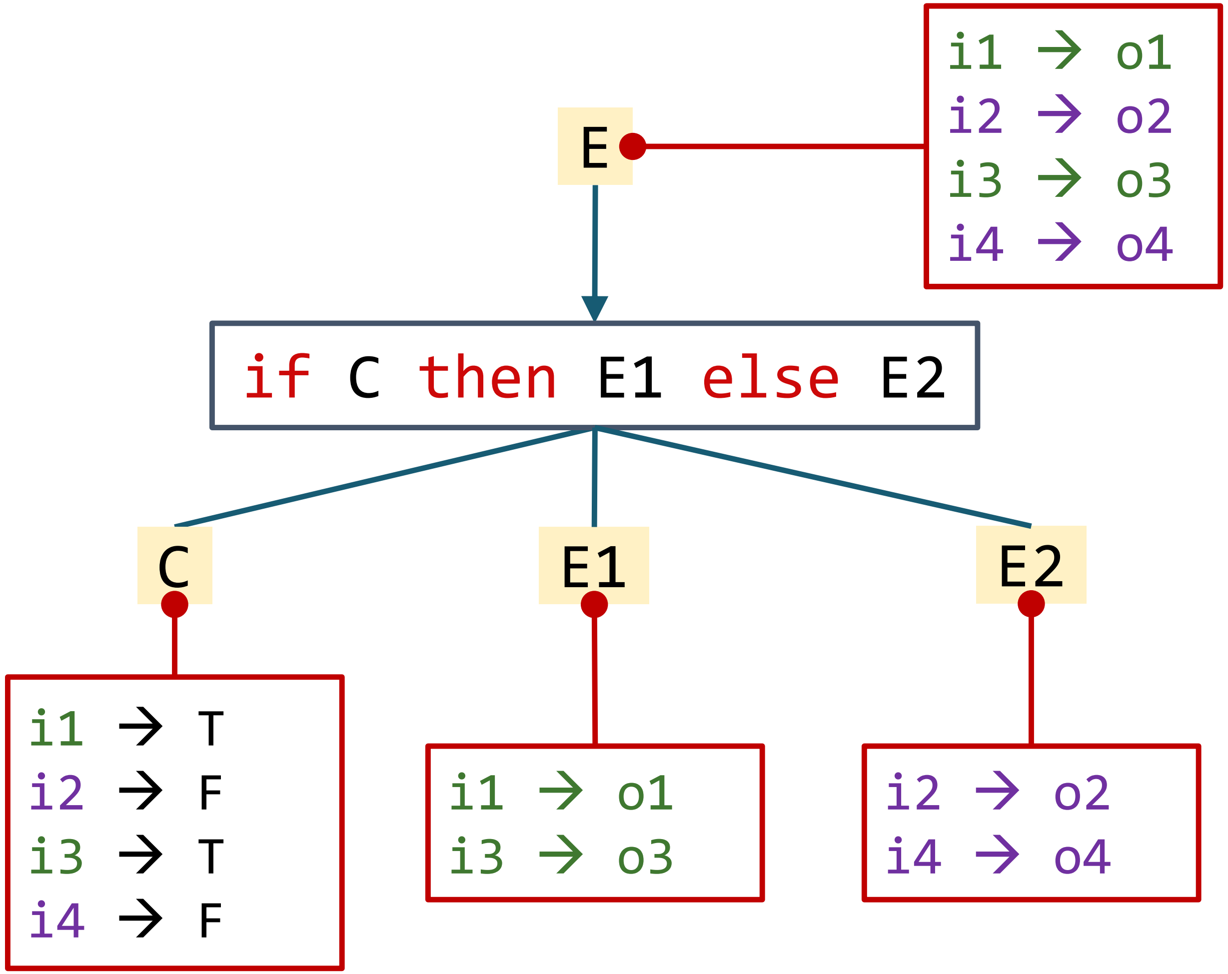
Smart way to synthesize conditionals

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?

EUSover

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

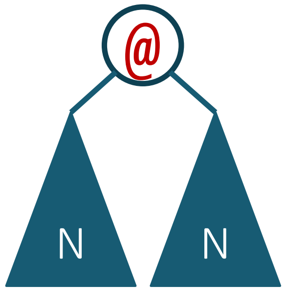
Q2: What are the main two pruning/decomposition techniques EUSolver uses to speed up the search? What enables these technique?

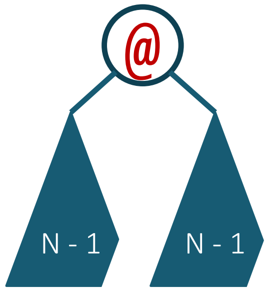
Q3: What would be a naive alternative to decision tree learning for synthesizing branch conditions? What are the disadvantages of this alternative?

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

Next week

Topics:

- Prioritization and Stochastic Search
- Representation-Based Synthesis

Paper: Gulwani: Automating string processing in spreadsheets using input-output examples

- Review due Wednesday
- Link to PDF on the course wiki
- Submit through EasyChair

Project: come talk to me about the topic!

- Monday 4-5pm