Lecture 3
Search Space Pruning

Nadia Polikarpova

Logistics

Reviews

- due tomorrow

Project

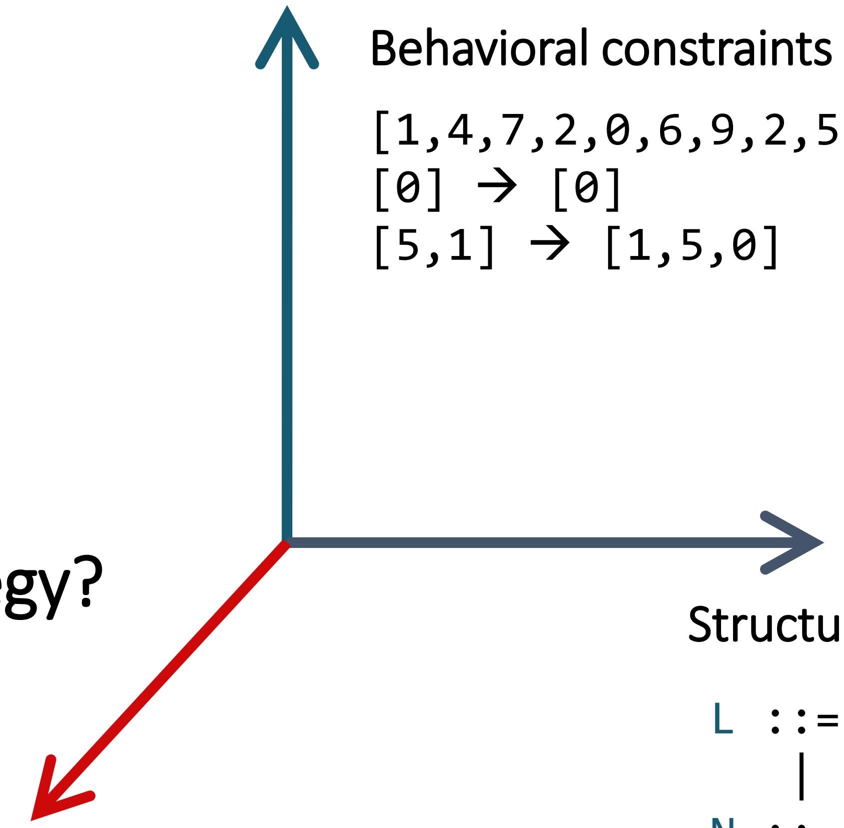
- teams due Friday
- if you haven’t found a team, sign up alone (I’ll merge singleton teams)

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]
```

Structural constraints = grammar

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

Search strategy?

Enumerative search

=

Explicit / Exhaustive Search

Idea: Enumerate 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]   ...
```

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 complete

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

Bottom-up

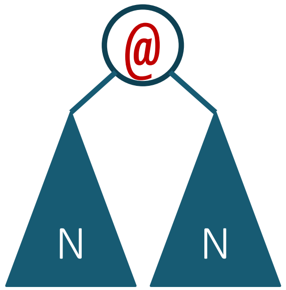
Candidates are complete but might not be whole

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

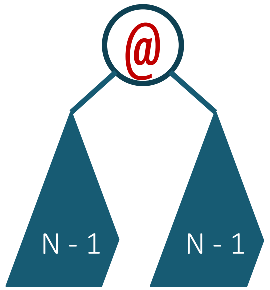
How to make it scale

Prune

Discard useless programs



$m * N^2$



$m * (N - 1)^2$

Prioritize

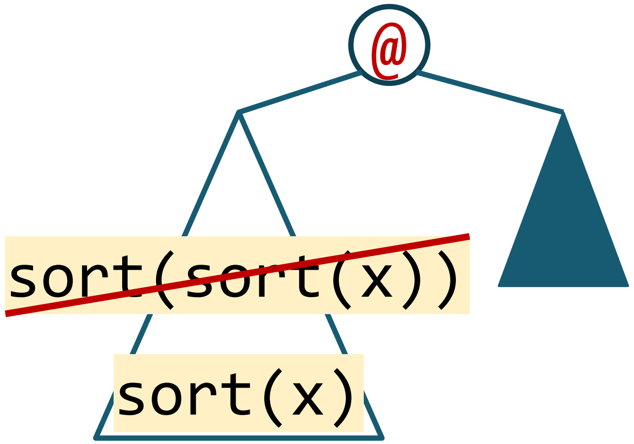
Explore promising candidates first

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

dequeue this first

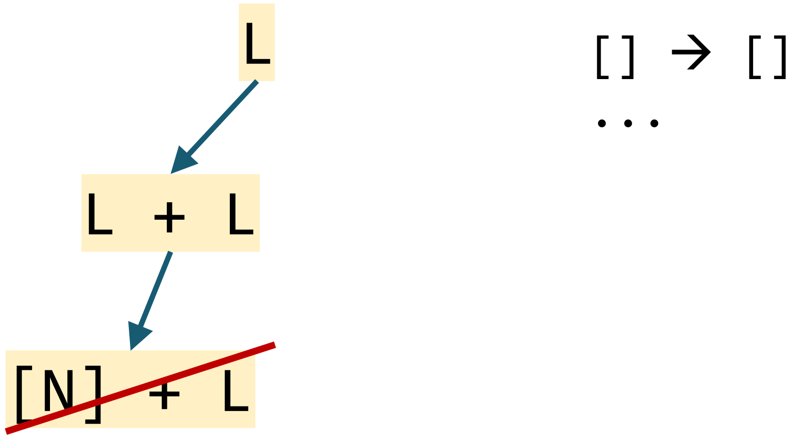
When can we discard a program?

redundant



Equivalence reduction

infeasible



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

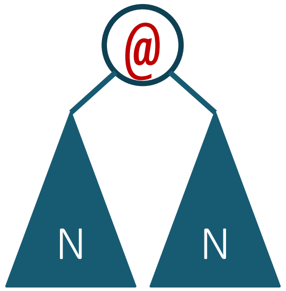
```
L ::= sort(L)     |
      L[N..N]     |
      L + L       |
      [N]         |
      x
N ::= find(L,N)   |
      0
```

bottom_up

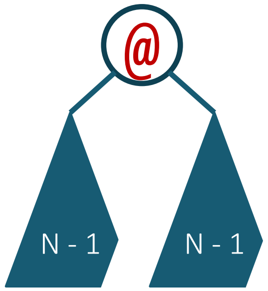
```
x   0

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 (<Σ, N, R, S>, [i → o]):
  bank[A,d] := {} forall A, d
  for d in [0..]:
    forall (A → rhs) in R:
      forall t in new-terms(A→rhs, d, bank):
        if (A = S ∧ t([i]) = [o]):
          return t
        if (forall t’ in bank[A,.]: !equiv(t,t’)):
          bank[A,d] += t

new-terms(A → σ(A₁…Aₖ), d, bank):
 if (d = 0 ∧ k = 0) yield σ
 else forall <d₁,…,dₖ> in [0..d-1]ᵏ s.t. max(d₁,…,dₖ) = d-1:
      forall <t₁,…,tₖ> in bank[A₁,d₁] × … × bank[Aₖ,dₖ]:
        yield σ(t₁,…,tₖ)
```



$m * N^2$



$m * (N - 1)^2$

Bottom-up + equivalence reduction

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank[A,d] := {} forall A, d
  for d in [0..]:
    forall (A → rhs) in R:
      forall t in new-terms(A→rhs, d, bank):
        if (A = S ∧ t([i]) = [o]):
          return t
        if (forall t' in bank[A,.]: !equiv(t,t')):
          bank[A,d] += t

new-terms(A → σ(A_1…A_k), d, bank):
 if (d = 0 ∧ k = 0) yield σ
 else forall <d_1,…,d_k> in [0..d-1]^k s.t. max(d_1,…,d_k) = d-1:
      forall <t_1,…,t_k> in bank[A_1,d_1] × … × bank[A_k,d_k]:
        yield σ(t_1,…,t_k)
```

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 (<Σ, N, R, S>, [i → o]):
{ ... }

equiv(t, t') {
  return t([i]) = t'([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 (<Σ, N, R, S>, [i → o]):
{ ... }

equiv(t, t') {
  return t([i]) = t'([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 (<Σ, N, R, S>, [i → o]):
{ ... }

equiv(t, t') {
  return t([i]) = t'([i])
}
```

```
[[0] → [0]]

x    0

x[0..0]    x + x

x + (x + x)
```

how to implement the reduction efficiently?

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)
- **EUSolver** [Alur et al. TACAS‘17]
- **Probe** [Barke et al. OOPSLA’20]
- **TFCoder** [Shi et al. TOPLAS’22]

User-specified 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` →$_{(n > m)}$ `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

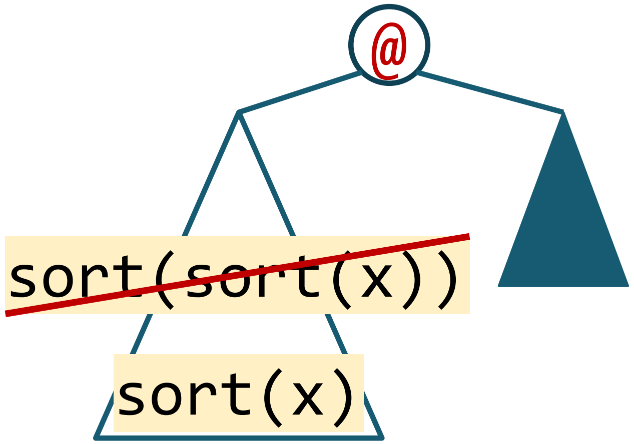
Built-in
- Even faster
- Restricted to built-in operators
- Only certain symmetries can be eliminated by modifying the grammar

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

Q2: Can any of them apply beyond PBE?

When can we discard a program?

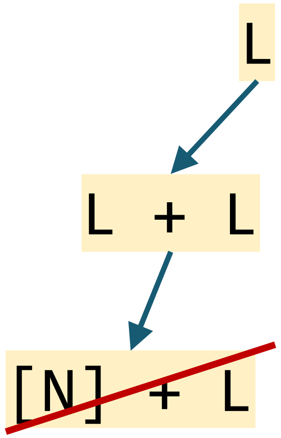
redundant



Equivalence reduction

(also: symmetry breaking)

infeasible



[] → []
...

Top-down propagation

Top-down search: reminder

generates a lot of incomplete terms
only discards complete 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]` `…..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:

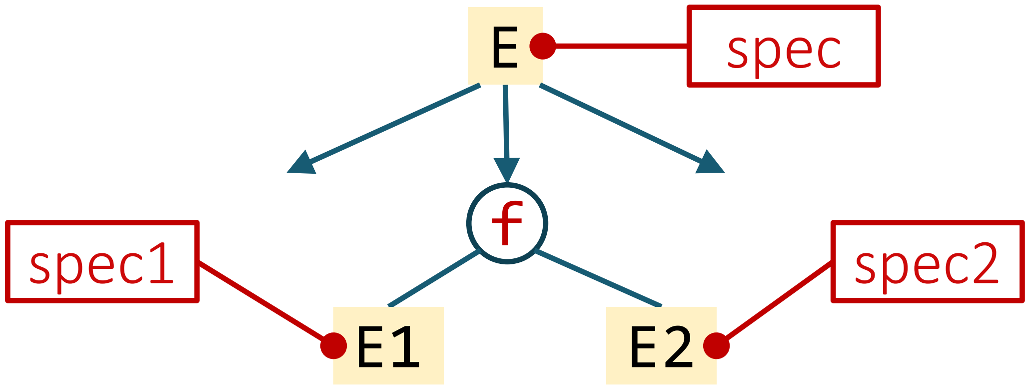
need to reject hopeless programs early!

```
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

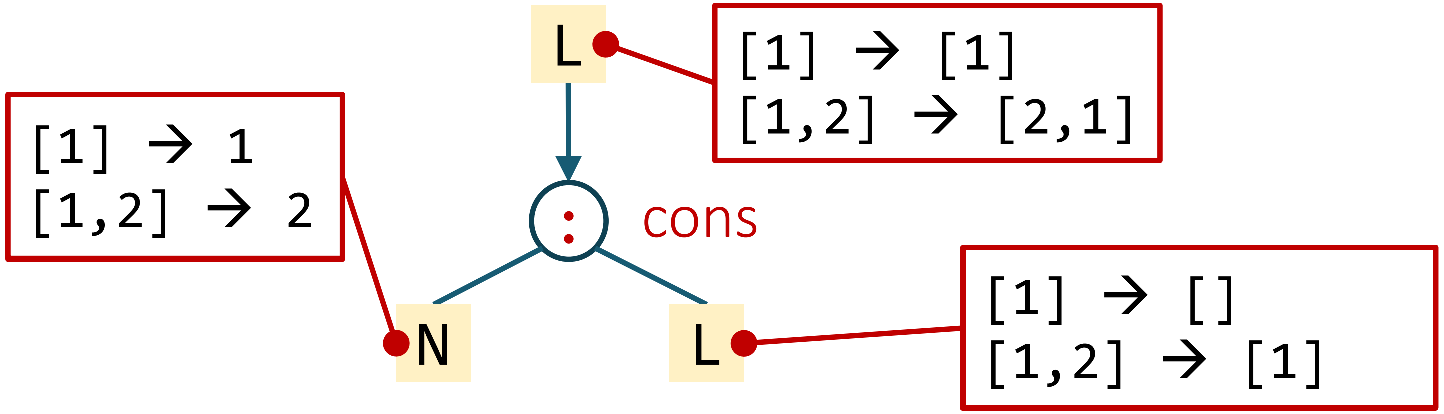


If `spec1 = ⊥` or `spec2 = ⊥` discard `f(E1,E2)`!

For now: spec = examples

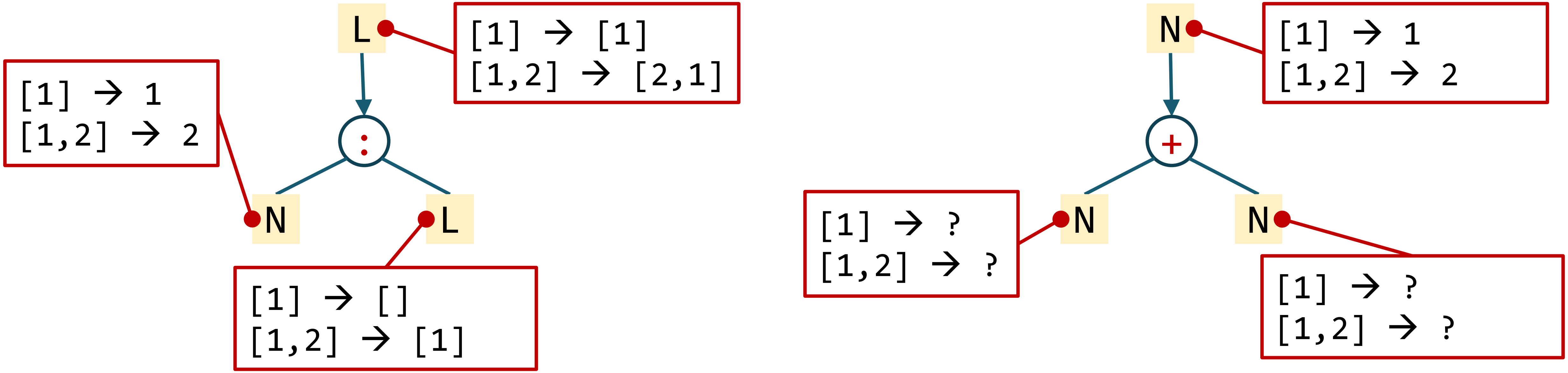
When is TDP possible?

Depends on f!



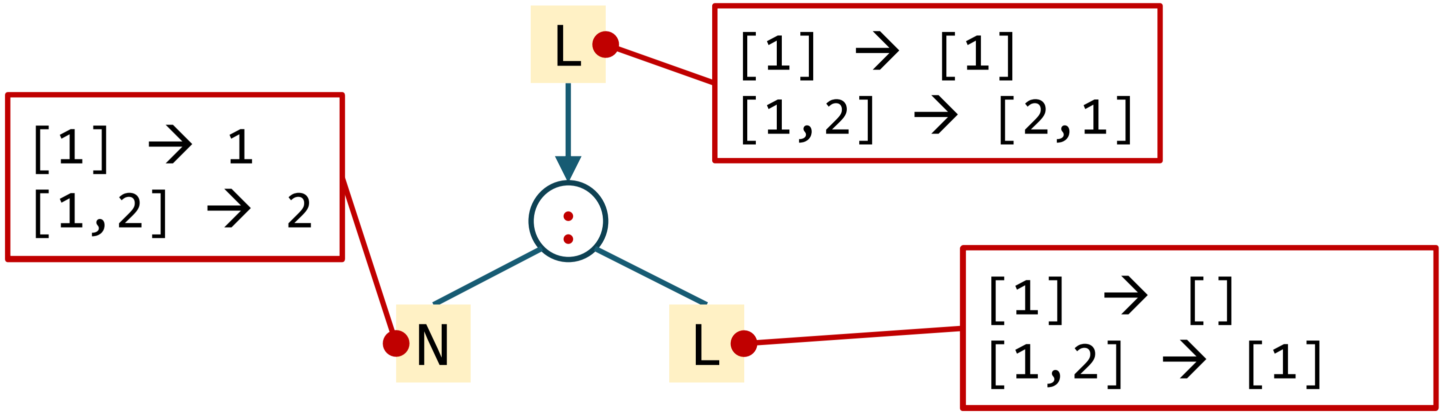
When is TDP possible?

Depends on f!



When is TDP possible?

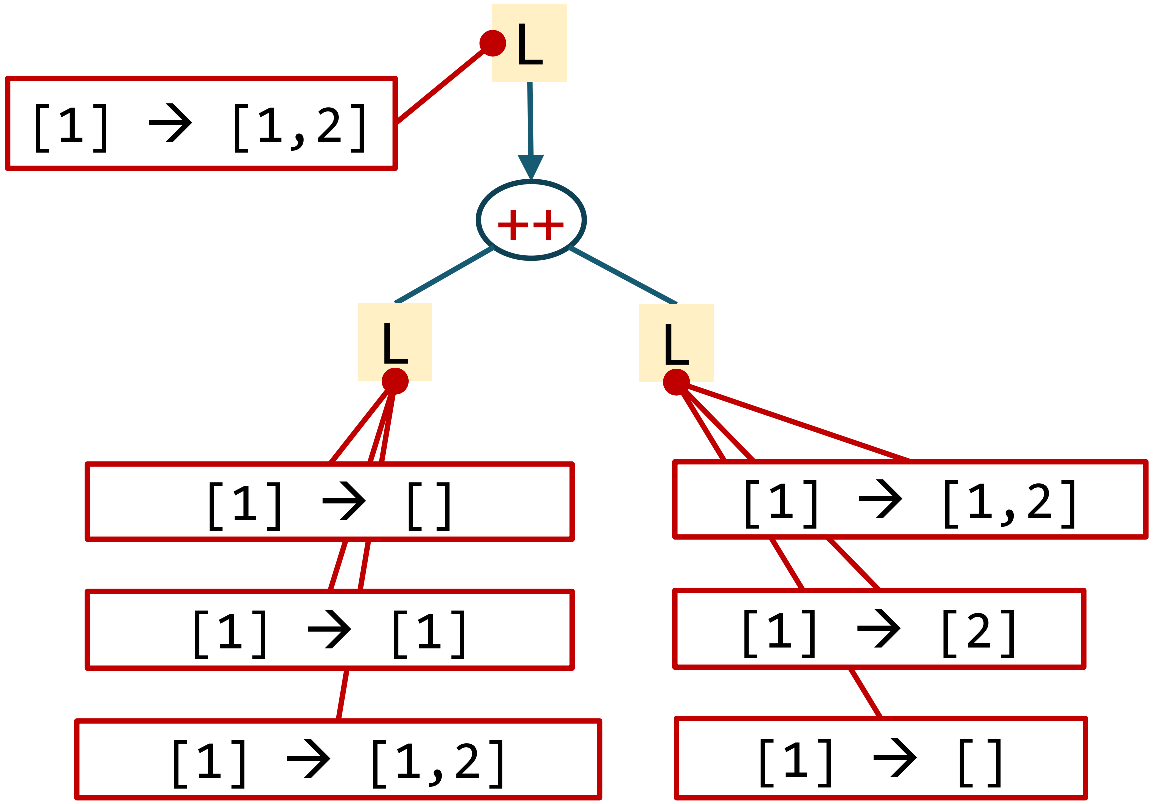
Depends on f!



Works when the function is injective!

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

Something in between?



Works when the function has a "small inverse"

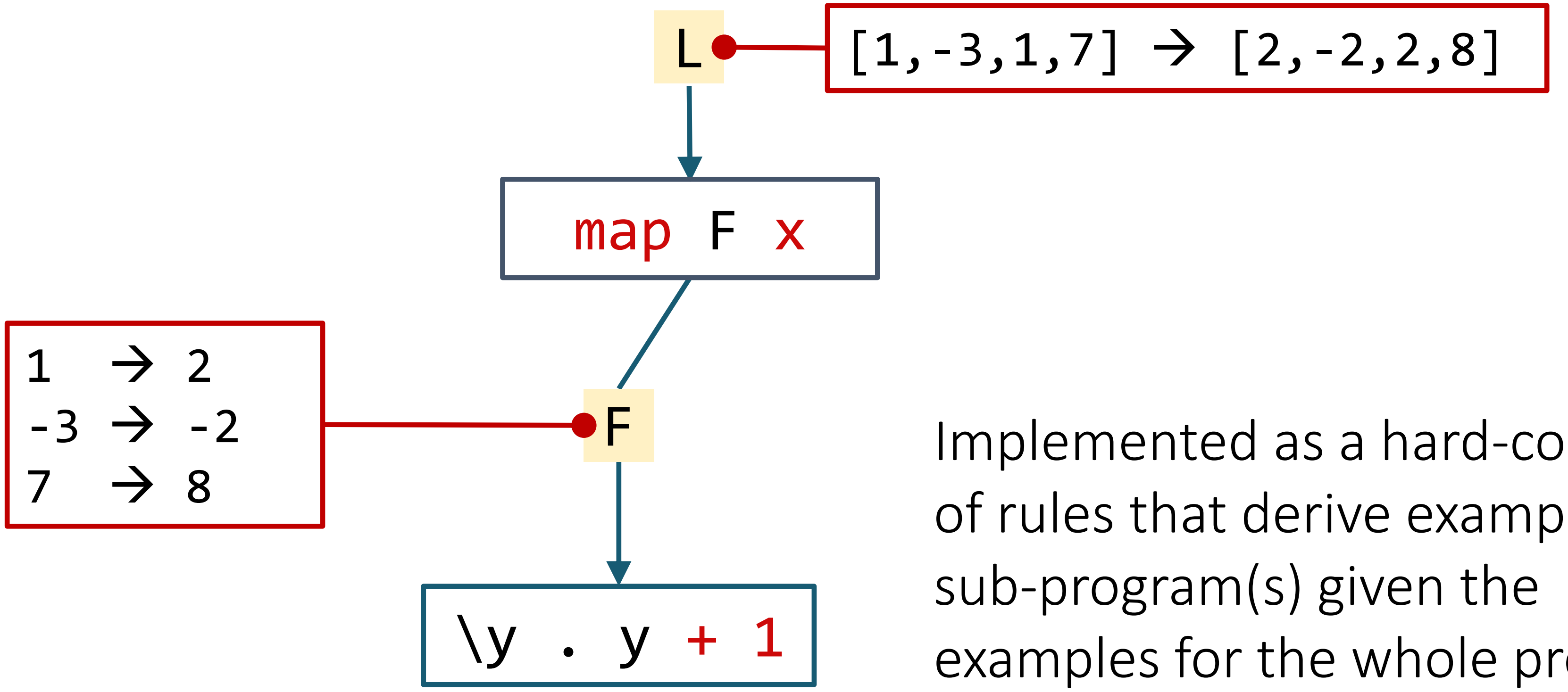
- or just the output examples have a small inverse

λ²: TDP for list combinators

[Feser, Chaudhuri, Dillig '15]

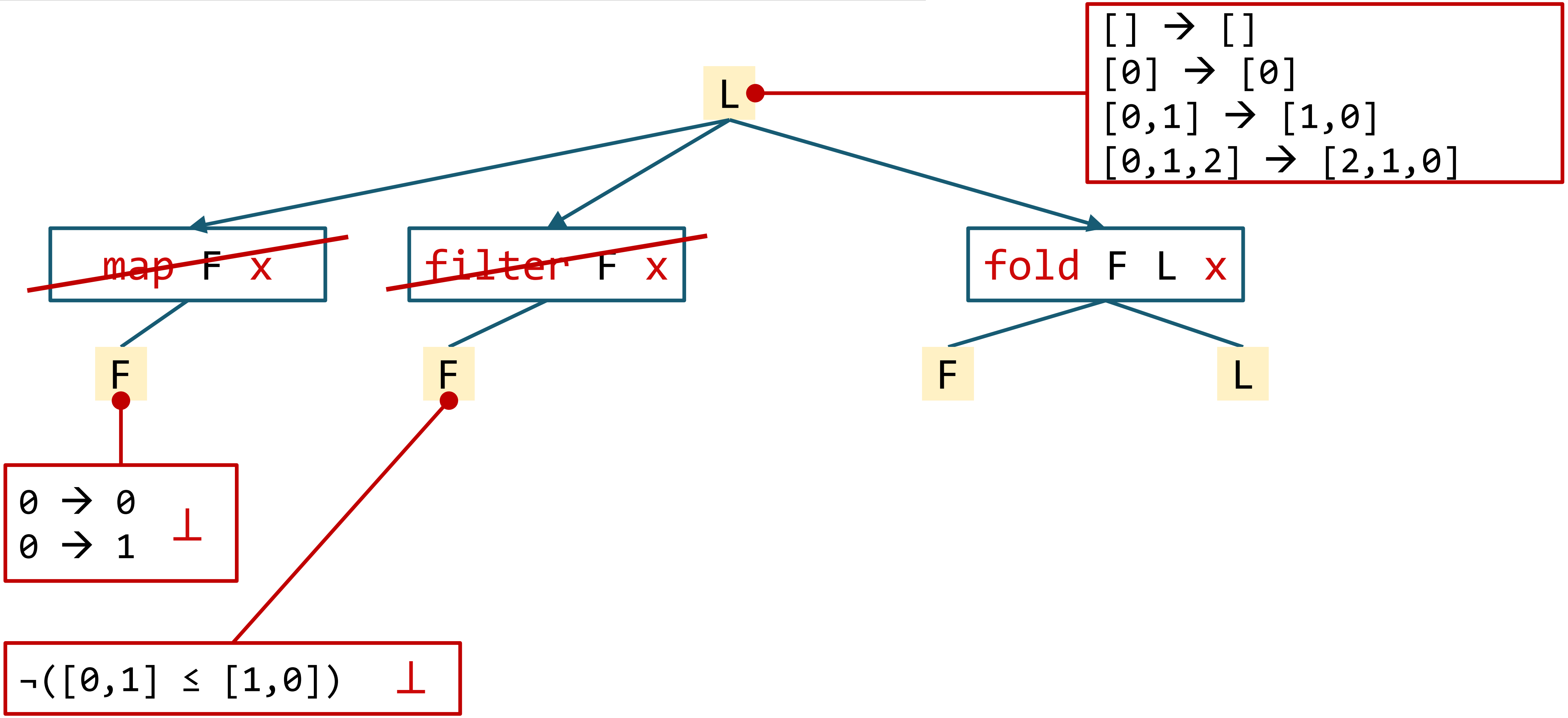
```
map f x             map (\y . y + 1) [1, -3, 1, 7] → [2, -2, 2, 8]

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

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

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

λ²: TDP for list combinators

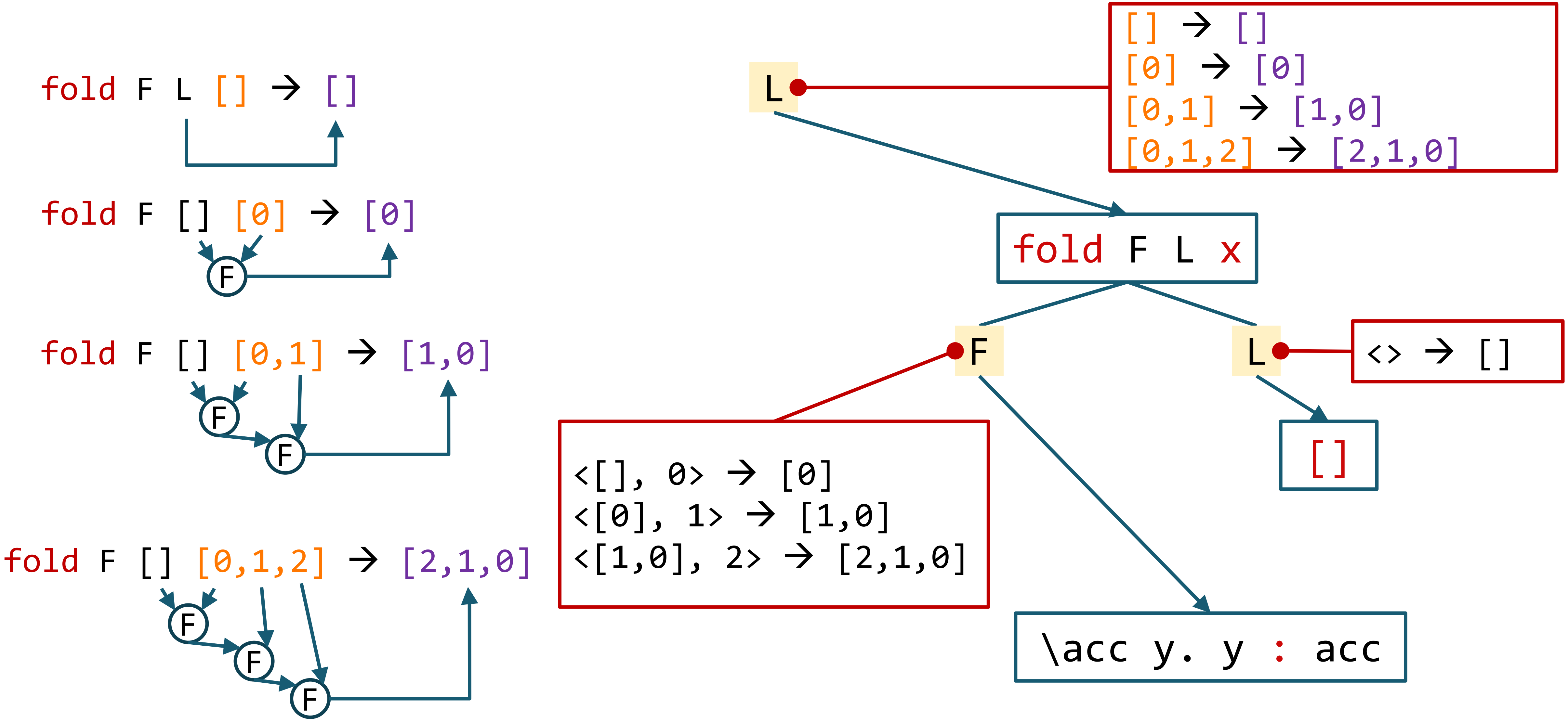


Implemented as a hard-coded set of rules that derive examples for sub-program(s) given the examples for the whole program and the combinator

λ²: TDP for list combinators



λ²: TDP for list combinators



Condition abduction

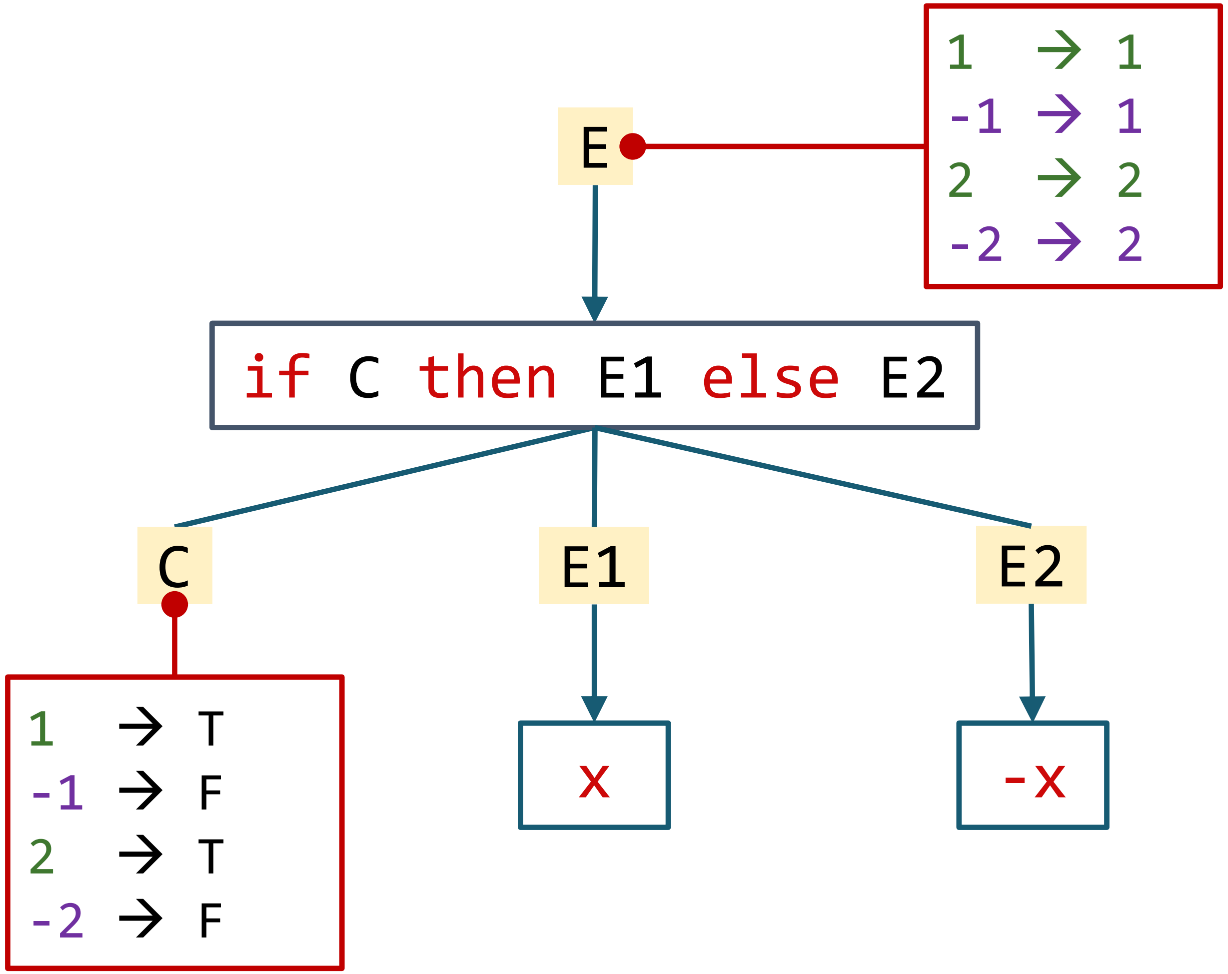
Smart way to synthesize conditionals

Used in many tools (under different names):

- **FlashFill** [Gulwani '11]
- **Escher** [Albarghouthi et al. '13]
- **Leon** [Kneuss et al. '13]
- **Synquid** [Polikarpova et al. '16]
- **EUSolver** [Alur et al. '17]

In fact, an instance of TDP!

Condition abduction



EUSolver

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

- First-order formula
- Conditional expression grammar
- Bottom-up enumerative with OE + 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

Empirically does well, especially on PBE

- why specifically on PBE?

EUSover: weaknesses

Only applies to conditional expressions

Does not always generate the smallest expression

- in the limit, can find the smallest solution
- but unclear when to stop

Only works for pointwise specifications

- but so do ALL CEGIS-based approaches

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.”

For strengths/weaknesses: use bullet points

Next week

Topics:

- Prioritizing/biasing the search

Paper: Lee, Heo, Alur, Naik: Accelerating Search-Based Program Synthesis using Learned Probabilistic Models. PLDI'18

- Review due Wednesday

Project:

- Proposals due in two weeks
- Talk to me about the topic