#6: Top-down Propagation

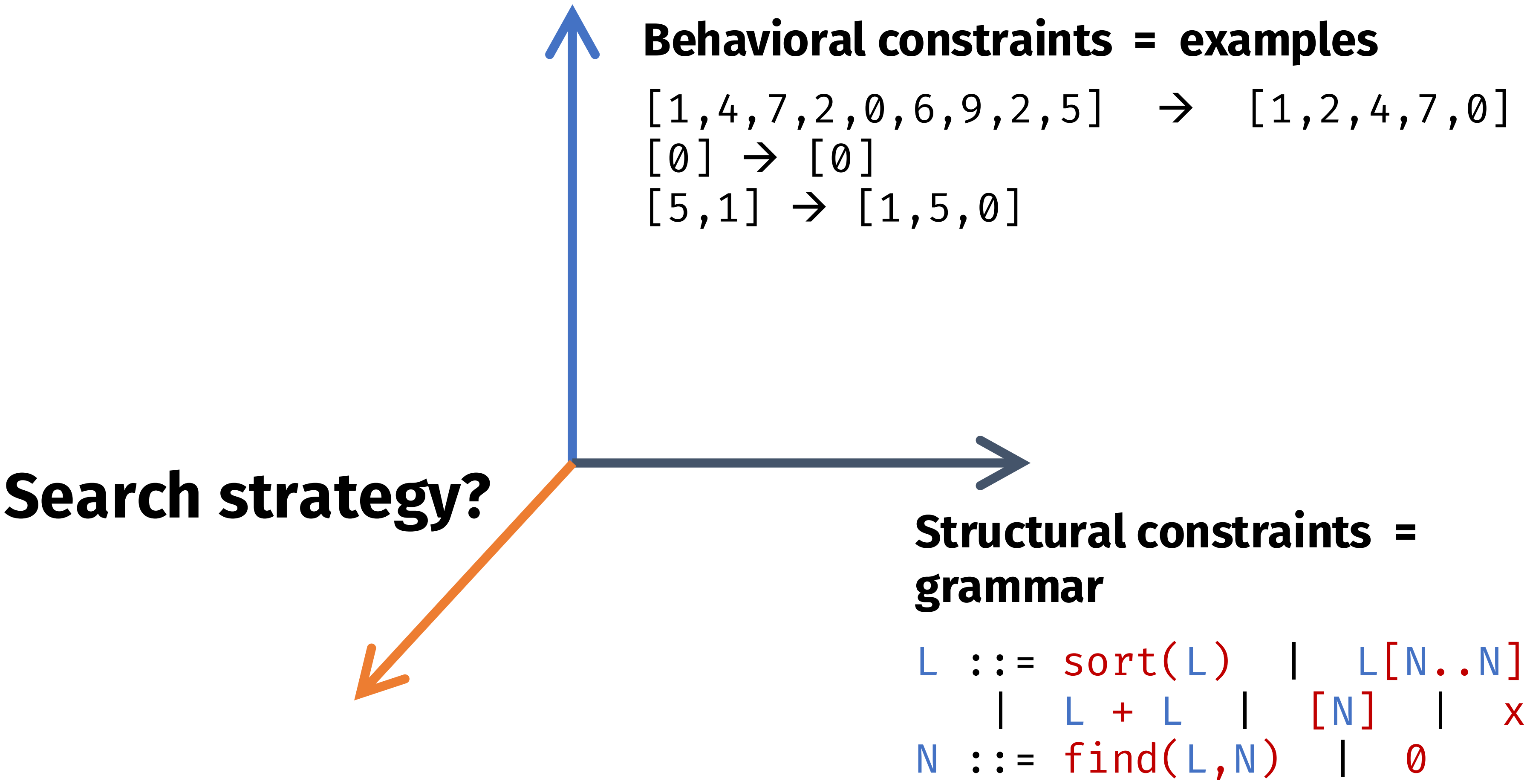
Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

Reminders

- Project Group due next week
- Start thinking about project
  - Meet with me!

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

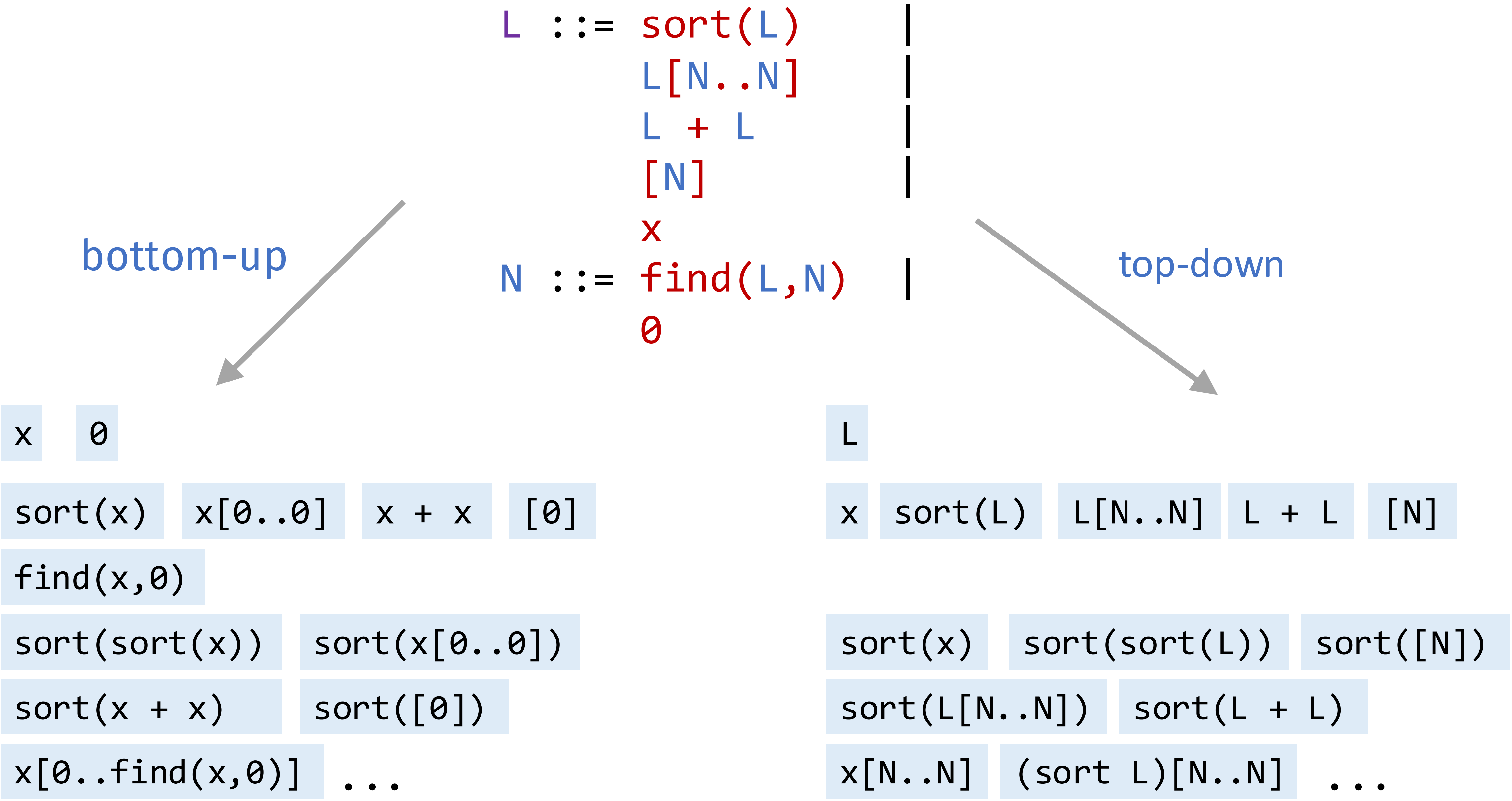
```
L ::= sort(L)  |  L[N..N]
   |  L + L  |  [N]  |  x
N ::= find(L,N)  |  0
```

Enumerative search

=

Explicit / Exhaustive Search

Idea: 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

top-down

```
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)]  ...
```

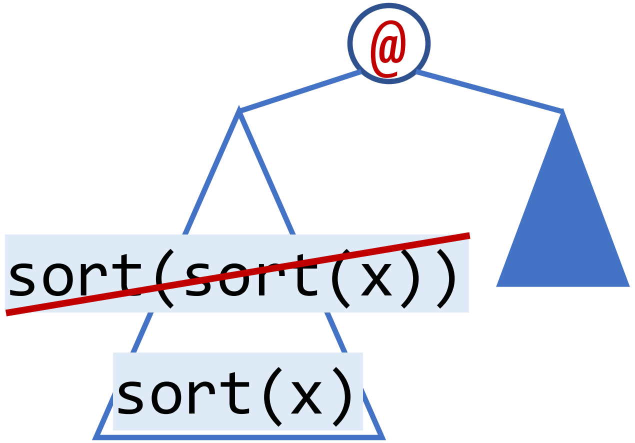
```
L

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]  ...
```

When can we discard a subprogram?

redundant



@

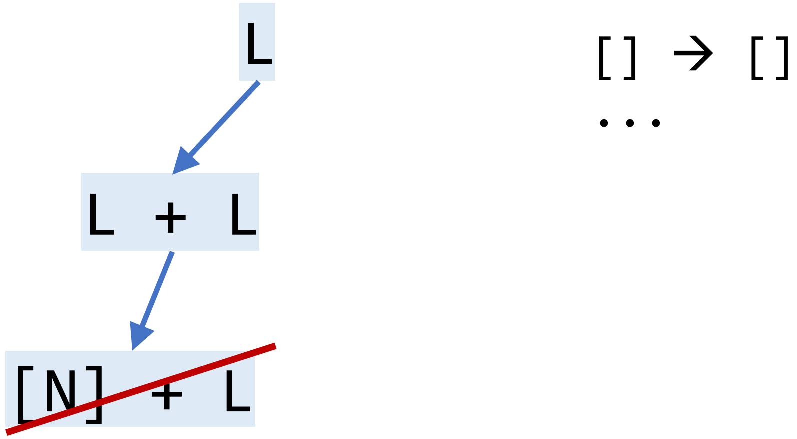
~~sort(sort(x))~~

sort(x)

Equivalence reduction

(also: symmetry breaking)

infeasible



L

L + L

~~[N] + L~~

[] → []

...

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:
```

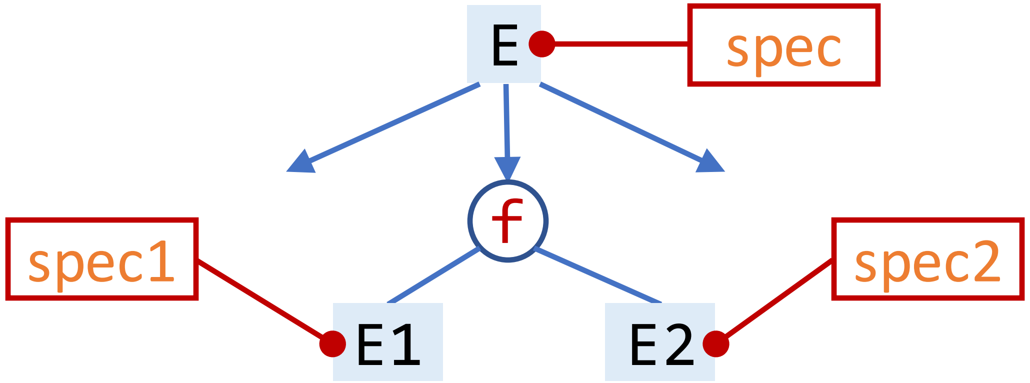
need to reject hopeless programs early!

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

```
[[1,4,0,6]  →  [1,4]]
```

Top-down propagation

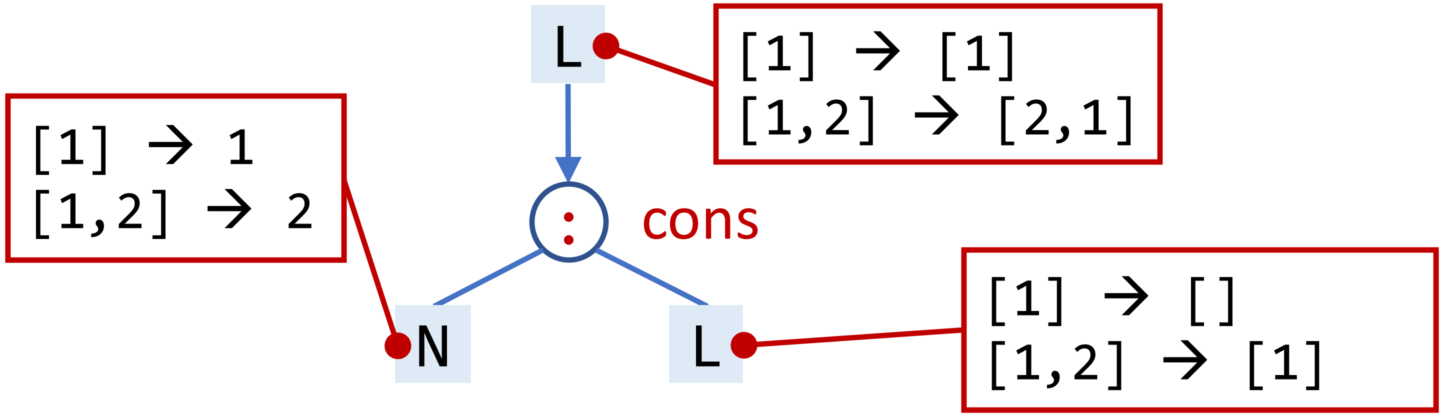
- **Idea:** once we pick the production, infer specs for subprograms



- If spec1 = ⊥ or spec2 = ⊥ discard f(E1,E2)!
- For now: spec = examples

When is TDP possible?

Depends on f!



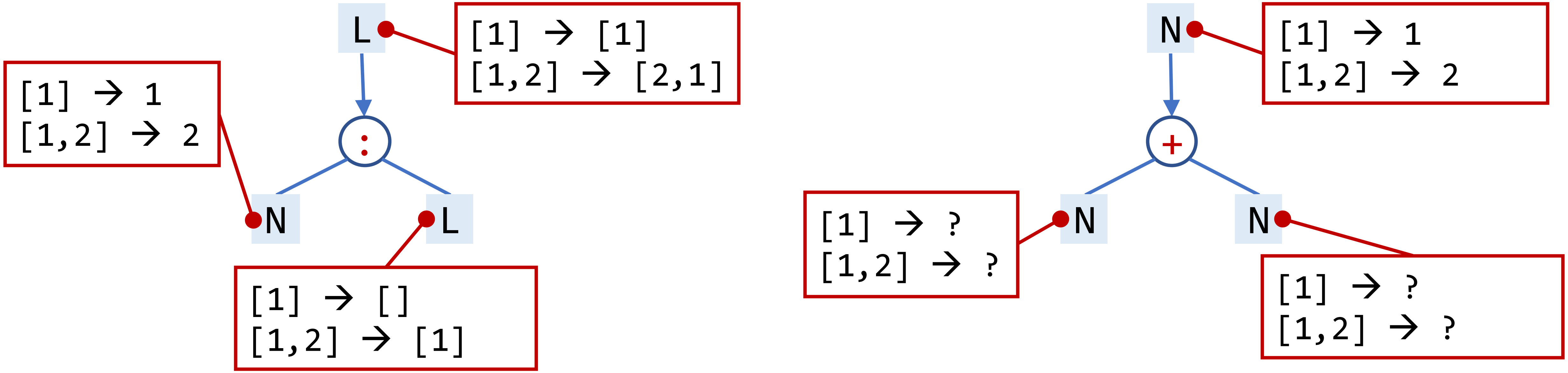
[1] → [1]
[1,2] → [2,1]

[1] → 1
[1,2] → 2

[1] → []
[1,2] → [1]

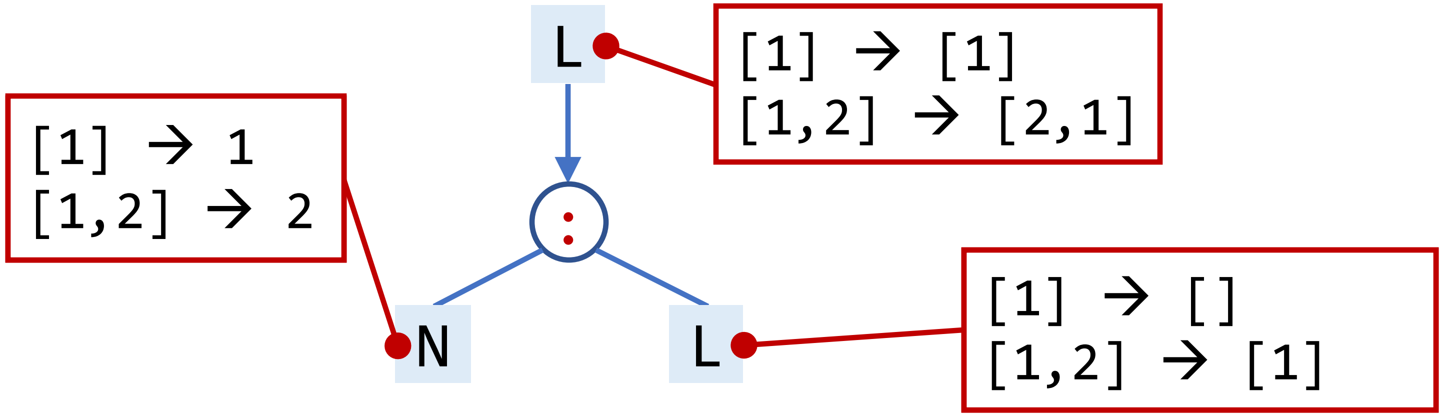
When is TDP possible?

Depends on f!



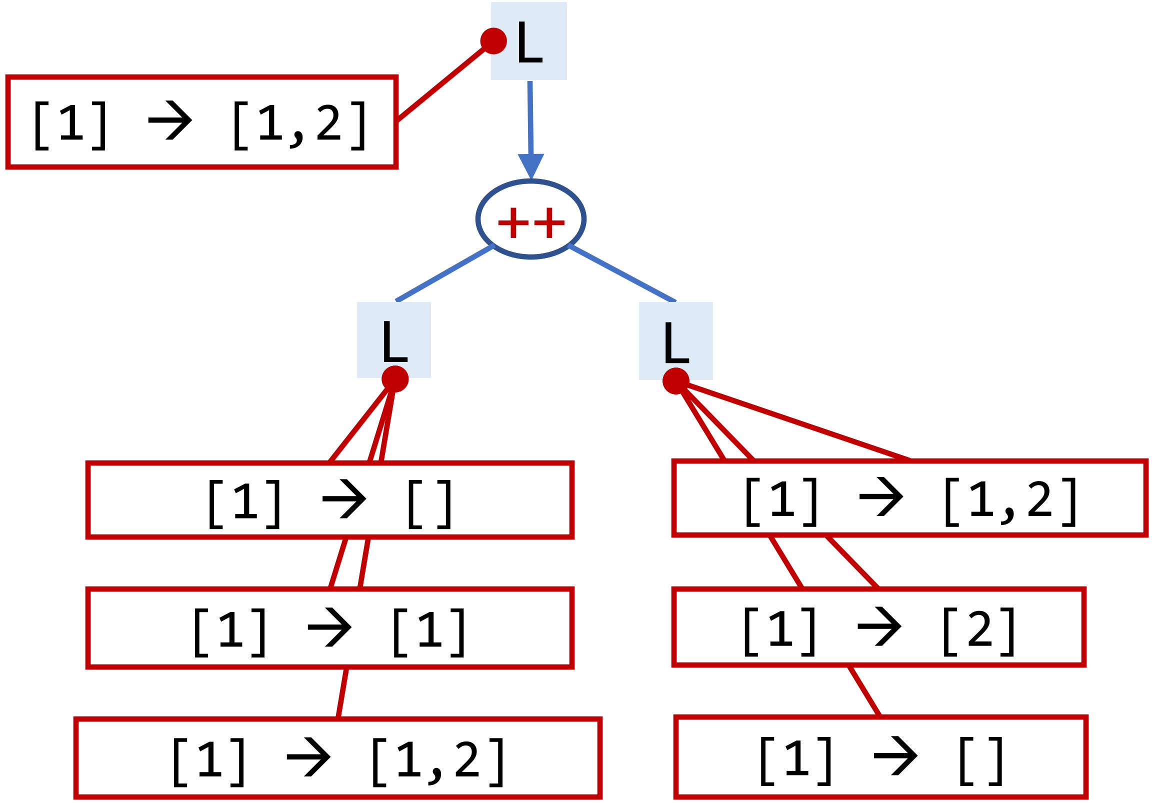
When is TDP possible?

- 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?

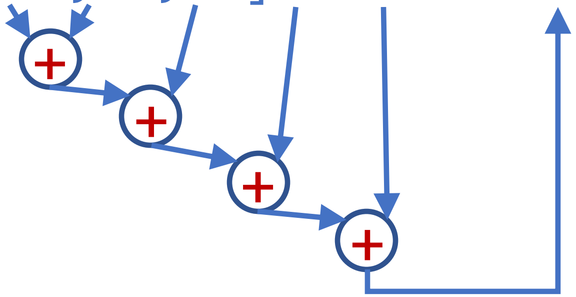


Works when the function has a "small inverse"

- or just the output examples have a small inverse

λ²: TDP for list combinators

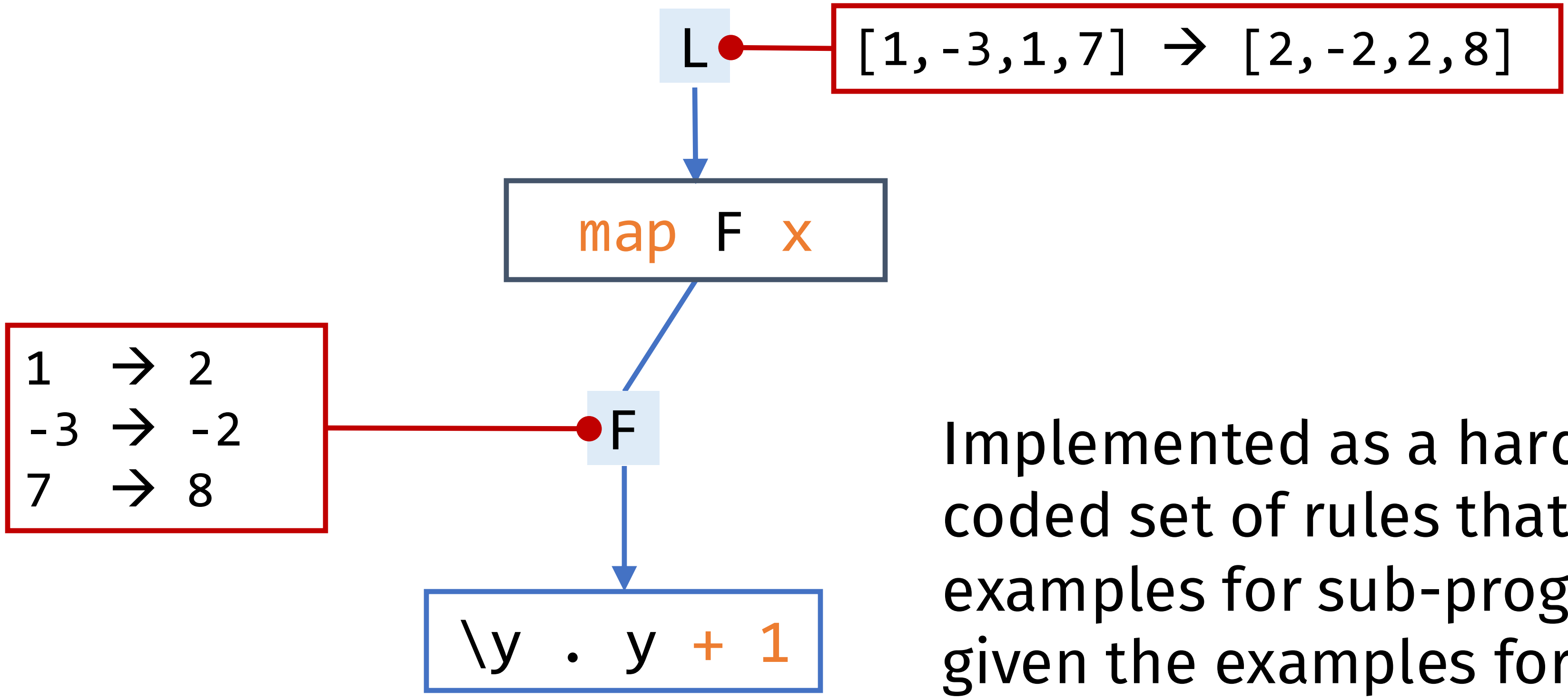
- `map f x` `map (\y . y + 1) [1, -3, 1, 7] → [2, -2, 2, 8]`
- `filter f x` `filter (\y . y > 0) [1, -3, 1, 7] → [1, 1, 7]`
- `fold f acc x` `fold (\y z . y + z) 0 [1, -3, 1, 7] → 6`



`fold (\y z . y + z) 0 [] → 0`

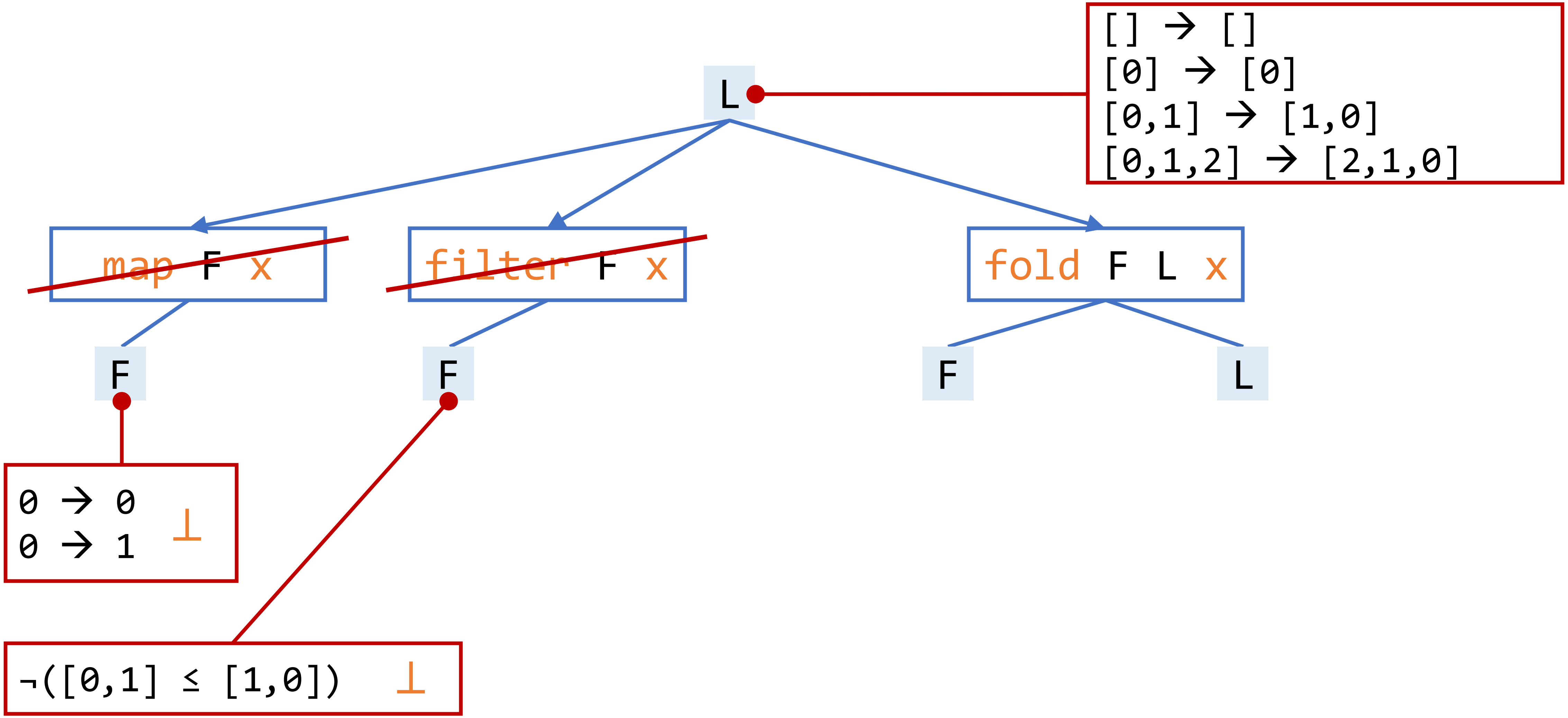
[Feser, Chaudhuri, Dillig ‘15]

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



L

[] → []
[0] → [0]
[0,1] → [1,0]
[0,1,2] → [2,1,0]

~~map F x~~

~~filter F x~~

fold F L x

F

F

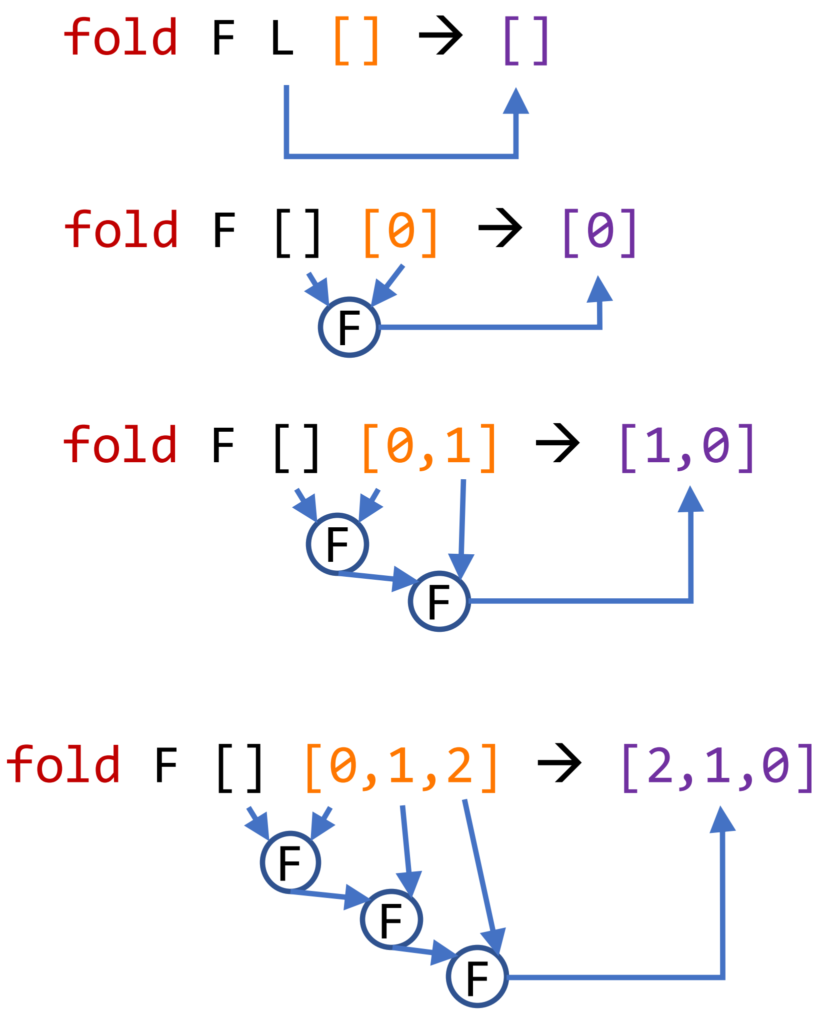
F

L

0 → 0
0 → 1 ⊥

¬([0,1] ≤ [1,0]) ⊥

λ²: TDP for list combinators

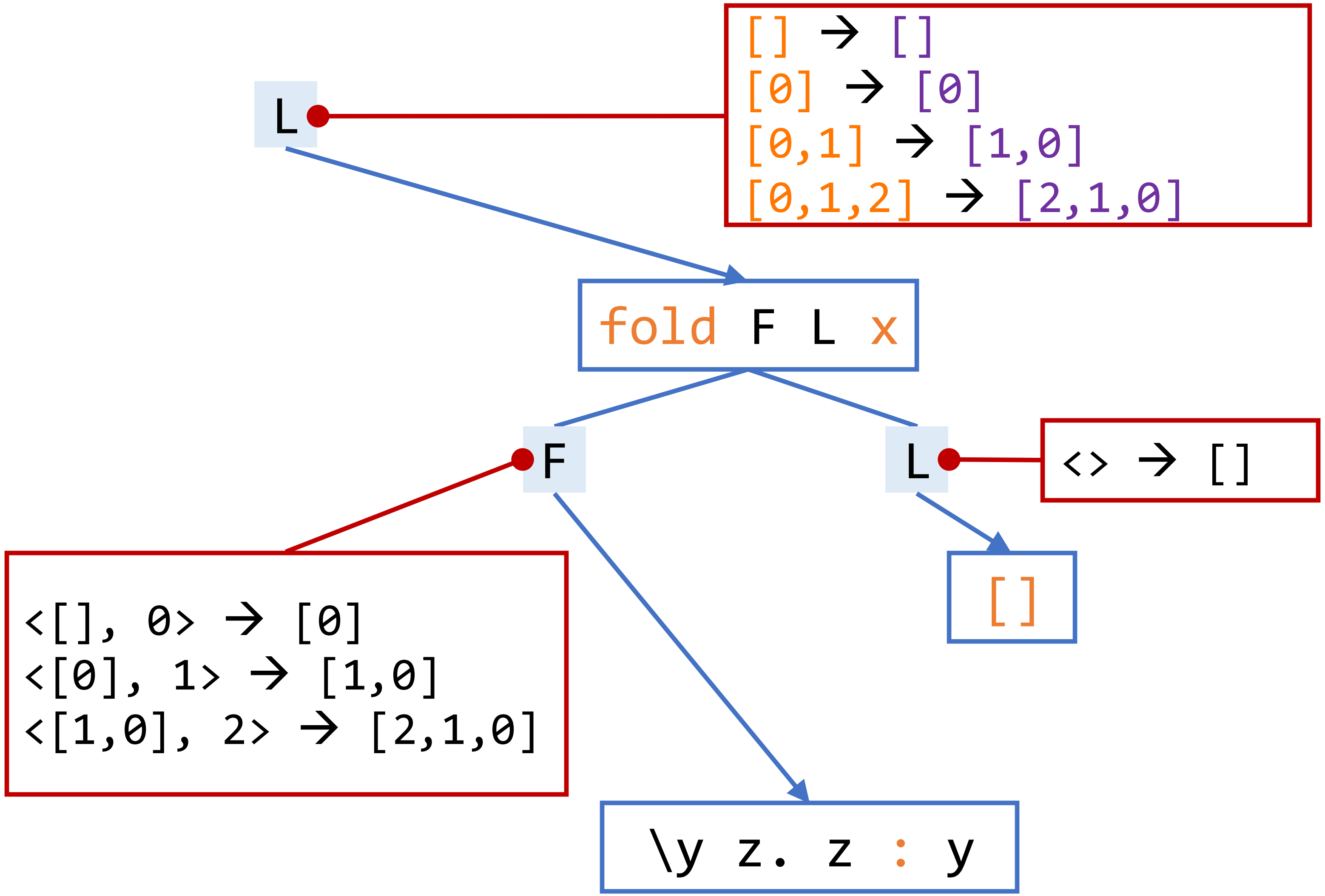


```
fold F L [] → []

fold F [] [0] → [0]

fold F [] [0,1] → [1,0]

fold F [] [0,1,2] → [2,1,0]
```



```
[] → []
[0] → [0]
[0,1] → [1,0]
[0,1,2] → [2,1,0]
```

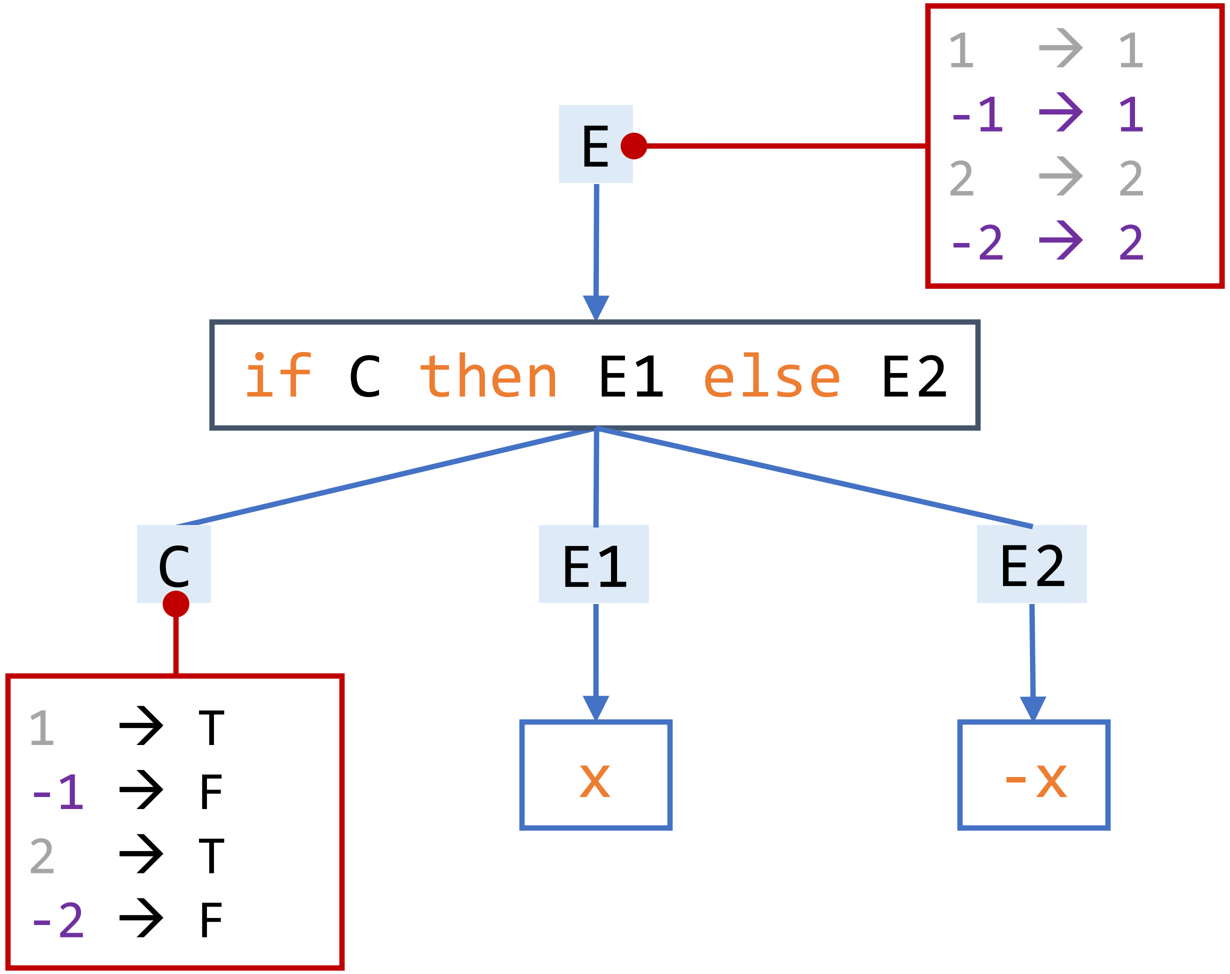
```
<[], 0> → [0]
<[0], 1> → [1,0]
<[1,0], 2> → [2,1,0]
```

```
<> → []
```

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



1	→	1
-1	→	1
2	→	2
-2	→	2

1	→	T
-1	→	F
2	→	T
-2	→	F