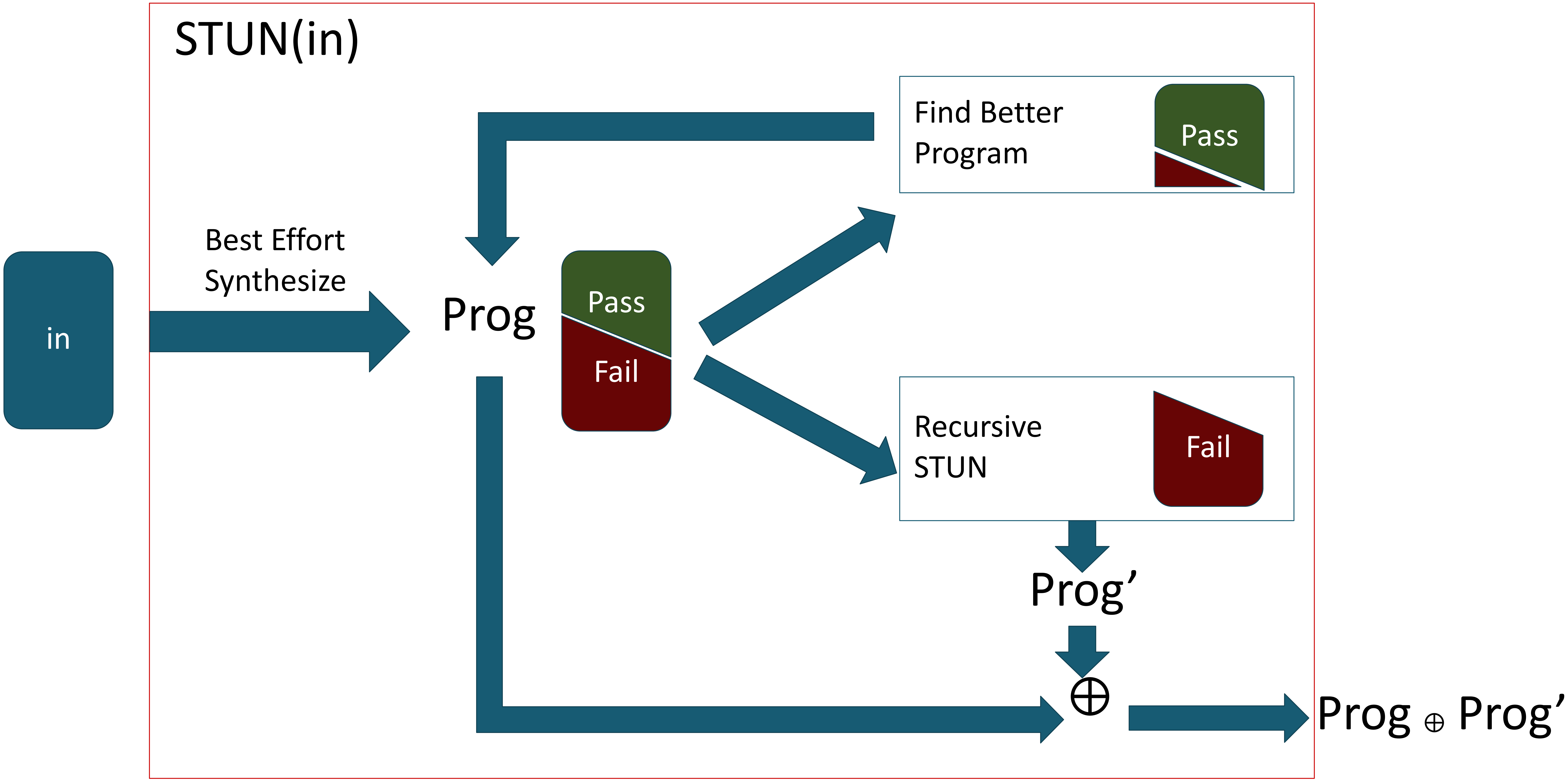
CS5733 Program Synthesis

#4. Other Search Techniques

Ashish Mishra

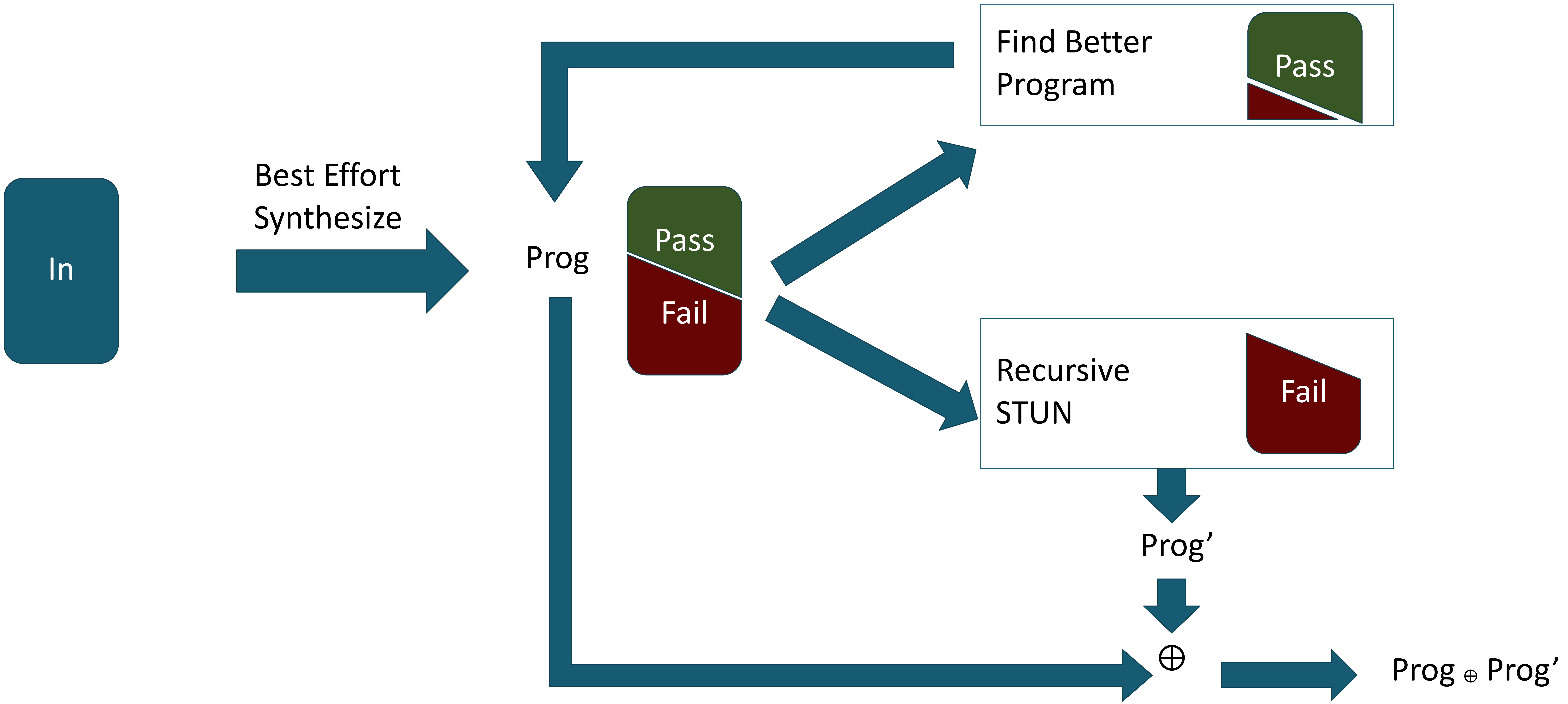
Revisit: Optimizing the Bottom-up Search

STUN at a glance



STUN at a glance

(1,2) -> 1
(2,3) -> 2
(4,3) -> 3
(8,1) -> 8



Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(8, 11)* -1 => -11
(3,6)*4 => 12
(-3, 8) *4 => -12

Grammar

```
expr = expr + expr
       | expr * expr
       | a | b | c
       | - expr
       if(bexp) expr else expr
bexp = expr > expr | expr > 0
```

Level 1:

a => [5,8,3,-3]	**b=>[10,11,6,8]**	**c=>[3,-1,4,4]**

Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(8, 11)* -1 => -11
(3,6)*4 => 12
(-3, 8) *4 => -12

Grammar

expr = expr + expr
| expr * expr
| a | b | c
| - expr
if(bexp) expr else expr
bexp = expr > expr| expr > 0

Level 1:

a => [5,8,3,-3]	b=>[10,11,6,8]	c=>[3,-1,4,4]

a+a=>[10,16,6,-6]	a+b=>[15,19,9,5]	a+c=>[8,7,7,1]	b+a=>[15,19,9,5]	b+b=>[20,22,12,16]	b+c=>[13,10,10,12]	c+a=>[8,7,7,1]
c+b=>[13,10,10,12]	c+c=>[6,-2,8,8]	a*a=>[25,64,9,9]	a*b=>[50,88,18,-48]	a*c=>[15,-8,12,-12]	b*a=>[50,88,18,-48]	b*b=>[100,121,36,64]
b*c=>[30,-11,24,32]	c*a=>[15,-8,12,-12]	c*b=>[30,-11,24,32]	c*c=>[9,1,16,16]	-a=>[-5,-8,-3,3]	-b=>[-10,-11,-6,-8]	-c=>[-3,1,-4,-4]

Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(8, 11)* -1 => -11
(3,6)*4 => 12
(-3, 8) *4 => -12

Eliminate Observationally equivalent ones

Grammar

expr = expr + expr
| expr * expr
| a | b | c
| - expr
|if(bexp) expr else expr
bexp = expr > expr | expr > 0

Level 1:

a => [5,8,3,-3]	b=>[10,11,6,8]	c=>[3,-1,4,4]

a+a=>[10,16,6,-6]	a+b=>[15,19,9,5]	a+c=>[8,7,7,1]	b+a=>[15,19,9,5]	b+b=>[20,22,12,16]	b+c=>[13,10,10,12]	c+a=>[8,7,7,1]
c+b=>[13,10,10,12]	c+c=>[6,-2,8,8]	a*a=>[25,64,9,9]	a*b=>[50,88,18,-48]	a*c=>[15,-8,12,-12]	b*a=>[50,88,18,-48]	b*b=>[100,121,36,64]
b*c=>[30,-11,24,32]	c*a=>[15,-8,12,-12]	c*b=>[30,-11,24,32]	c*c=>[9,1,16,16]	-a=>[-5,-8,-3,3]	-b=>[-10,-11,-6,-8]	-c=>[-3,1,-4,-4]

Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(8, 11)* -1 => -11
(3,6)*4 => 12
(-3, 8) *4 => -12

Identify an expression that works for a subset of the inputs

Grammar

expr = expr + expr
| expr * expr
| a | b | c
| - expr
| if(bexp) expr else expr
bexp = expr > expr | expr > 0

Level 1:

a => [5,8,3,-3]	b=>[10,11,6,8]	c=>[3,-1,4,4]

a+a=>[10,16,6,-6]	a+b=>[15,19,9,5]	a+c=>[8,7,7,1]	b+a=>[15,19,9,5]	b+b=>[20,22,12,16]	b+c=>[13,10,10,12]	c+a=>[8,7,7,1]
c+b=>[13,10,10,12]	c+c=>[6,-2,8,8]	a*a=>[25,64,9,9]	a*b=>[50,88,18,-48]	a*c=>[15,-8,12,-12]	b*a=>[50,88,18,-48]	b*b=>[100,121,36,64]
b*c=>[30,-11,24,32]	c*a=>[15,-8,12,-12]	c*b=>[30,-11,24,32]	c*c=>[9,1,16,16]	-a=>[-5,-8,-3,3]	-b=>[-10,-11,-6,-8]	-c=>[-3,1,-4,-4]

Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(8, 11)* -1 => -11
(3,6)*4 => 12
(-3, 8) *4 => -12

Identify an expression that works for the rest of the inputs

Grammar

expr = expr + expr
| expr * expr
| a | b | c
| - expr
| if(bexp) expr else expr
bexp = expr > expr| expr > 0

Level 1:

a => [5,8,3,-3]	b=>[10,11,6,8]	c=>[3,-1,4,4]

a+a=>[10,16,6,-6]	a+b=>[15,19,9,5]	a+c=>[8,7,7,1]	b+a=>[15,19,9,5]	b+b=>[20,22,12,16]	b+c=>[13,10,10,12]	c+a=>[8,7,7,1]
c+b=>[13,10,10,12]	c+c=>[6,-2,8,8]	a*a=>[25,64,9,9]	a*b=>[50,88,18,-48]	a*c=>[15,-8,12,-12]	b*a=>[50,88,18,-48]	b*b=> [100,121,36,64]
b*c=>[30,-11,24,32]	c*a=>[15,-8,12,-12]	c*b=>[30,-11,24,32]	c*c=>[9,1,16,16]	-a=>[-5,-8,-3,3]	-b=>[-10,-11,-6,-8]	-c=>[-3,1,-4,-4]

Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(3,6)*4 => 12
(-3, 8) *4 => -12

a*c P_1

(8, 11)* -1 => -11

b*c P_2

Grammar

expr = expr + expr
| expr * expr
| a | b | c
| - expr
| if(bexp) expr else expr
bexp = expr > expr| expr > 0

a>a=[f,f,f,f]	a>b=[f,f,f,f]	a>c=[t,f,f,t]
b>a=[t,t,t,t]	b>b=[f,f,f,f]	b>c=[t,t,t,t]
c>a=[f,t,t,f]	c>b=[f,f,f,f]	c>c=[f,f,f,f]
a>0=[t,t,f,t]	b>0=[t,t,t,t]	c>0=[t,t,t,f]

Simple case: (a,b) * c => out

Examples

(5,10) * 3 => 15
(3,6)*4 => 12
(-3, 8) *4 => -12

a*c P_1

(8, 11)* -1 => -11

b*c P_2

Grammar

expr = expr + expr
| expr * expr
| a | b | c
| - expr
| if(bexp) expr else expr
bexp = expr > expr| expr > 0

a>a=[f,f,f,f]	a>b=[f,f,f,f]	a>c=[t,f,f,t]
b>a=[t,t,t,t]	b>b=[f,f,f,f]	b>c=[t,t,t,t]
c>a=[f,t,t,f]	c>b=[f,f,f,f]	c>c=[f,f,f,f]
a>0=[t,t,f,t]	b>0=[t,t,t,t]	c>0=[t,t,t,f]

$$P_1 \oplus P_2 = if(c > 0)\ a * c\ else\ \ b * c$$

Another approach: Hierarchical Search

- When can we separate a problem into simpler subproblems?
  - What if separating based on input examples is infeasible?
  - Chenglong Wang, Alvin Cheung, Rastislav Bodik, Synthesizing Highly Expressive SQL Queries from Input-output Examples, 2017.
  - Key insight: the problem can be decomposed in a hierarchical way.

Example: SQL

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Output

Output

XX

Todd
Sally

Language

```
Rel      := T | Rel , Rel
              | Select Fields  from Rel where Pred

Pred     := exp = exp  |  exp > exp |  Pred & Pred

Fields   :=  table.name as name | table.name as name, Fields
```

Hierarchical Search

- Key idea:
  - First search for the *structure* of the query
  - Then search for the details of the predicates

These structures are also called Hypothesis space.

- Observation:
  - If a query has the wrong structure we can see it has the wrong structure before instantiating the details

Language with holes

The key idea is to define a semantics for queries with holes that is guaranteed to produce a superset of the records that any instantiation of the holes may produce

```
Rel     := T | Rel , Rel
             | Select Fields  from Rel where □

Fields  :=  table.name as name  |  table.name as name,
Fields
```

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Query

Superset of output

Language with holes

```
Rel     := T | Rel , Rel
             | Select Fields  from Rel where □

Fields  :=  table.name as name  |  table.name as name,
Fields
```

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Query

Employee

Superset of output

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Language with holes

```
Rel     := T | Rel , Rel
             | Select Fields  from Rel where □

Fields  :=  table.name as name  |  table.name as name,
Fields
```

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Query

Depts

Superset of output

Sales, A1
Engineering, A2
Operations, A1

Language with holes

```
Rel     := T | Rel , Rel
             | Select Fields  from Rel where □

Fields  :=  table.name as name  |  table.name as name,
Fields
```

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Query

Employee, Depts

Superset of output

Todd, Sales, Sales, A1
Todd, Sales, Engineering, A2
Todd, Sales, Operations, A1
Joe, Engineering, Sales, A1
Joe, Engineering, Engineering, A2
Joe, Engineering, Operations, A1
Alice, Engineering, Sales, A1
Alice, Engineering, Engineering, A2
Alice, Engineering, Engineering, A3
Sally, Operations, Sales, A1
Sally, Operations, Engineering, A2
Sally, Operations, Operations, A1

Language with holes

```
Rel     := T | Rel , Rel
             | Select Fields  from Rel where □

Fields  :=  table.name as name   |  table.name as name,
Fields
```

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Query

Select Name from Employee
where □

Superset of output

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Language with holes

```
Rel     := T | Rel , Rel
             | Select Fields  from Rel where □

Fields  :=  table.name as name  |  table.name as name,
Fields
```

Input

Employee

Name, Dept

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Depts

Dept, Building

Sales, A1
Engineering, A2
Operations, A1

Query

Select Name **from**
Employee, Depts
where □

Superset of output

Todd, Sales, Sales, A1
Todd, Sales, Engineering, A2
Todd, Sales, Operations, A1
Joe, Engineering, Sales, A1
Joe, Engineering, Engineering, A2
Joe, Engineering, Operations, A1
Alice, Engineering, Sales, A1
Alice, Engineering, Engineering, A2
Alice, Engineering, Engineering, A3
Sally, Operations, Sales, A1
Sally, Operations, Engineering, A2
Sally, Operations, Operations, A1

Viable Queries

Select Name from Employee

where □

Todd, Sales
Joe, Engineering
Alice, Engineering
Sally, Operations

Can we find the right predicate?

This is an inductive synthesis problem too!

Select Name **from**
Employee, Depts
where □

Todd, Sales, Sales, A1
Todd, Sales, Engineering, A2
Todd, Sales, Operations, A1
Joe, Engineering, Sales, A1
Joe, Engineering, Engineering, A2
Joe, Engineering, Operations, A1
Alice, Engineering, Sales, A1
Alice, Engineering, Engineering, A2
Alice, Engineering, Engineering, A3
Sally, Operations, Sales, A1
Sally, Operations, Engineering, A2
Sally, Operations, Operations, A1

Viable Queries

Select Name **from**
Employee, Depts
where □

Todd, Sales, Sales, A1
Todd, Sales, Engineering, A2
Todd, Sales, Operations, A1
Joe, Engineering, Sales, A1
Joe, Engineering, Engineering, A2
Joe, Engineering, Operations, A1
Alice, Engineering, Sales, A1
Alice, Engineering, Engineering, A2
Alice, Engineering, Engineering, A3
Sally, Operations, Sales, A1
Sally, Operations, Engineering, A2
Sally, Operations, Operations, A1

Employee.Dept = Depts.Dept
& Dept=A1

Todd, Sales, Sales, A1
Sally, Operations, Operations, A1

Pruning in Top-down enumeration using specs

=

Top-down Propagation

Top-down vs Bottom-up: Basic Philosophy

Guiding the enumeration + Pruning using Outputs

Guiding the enumeration + Pruning using Inputs

Top-down search: reminder

Worklist w

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: example (depth-first)

Worklist w

Generates a lot of incomplete terms while only discards complete terms

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

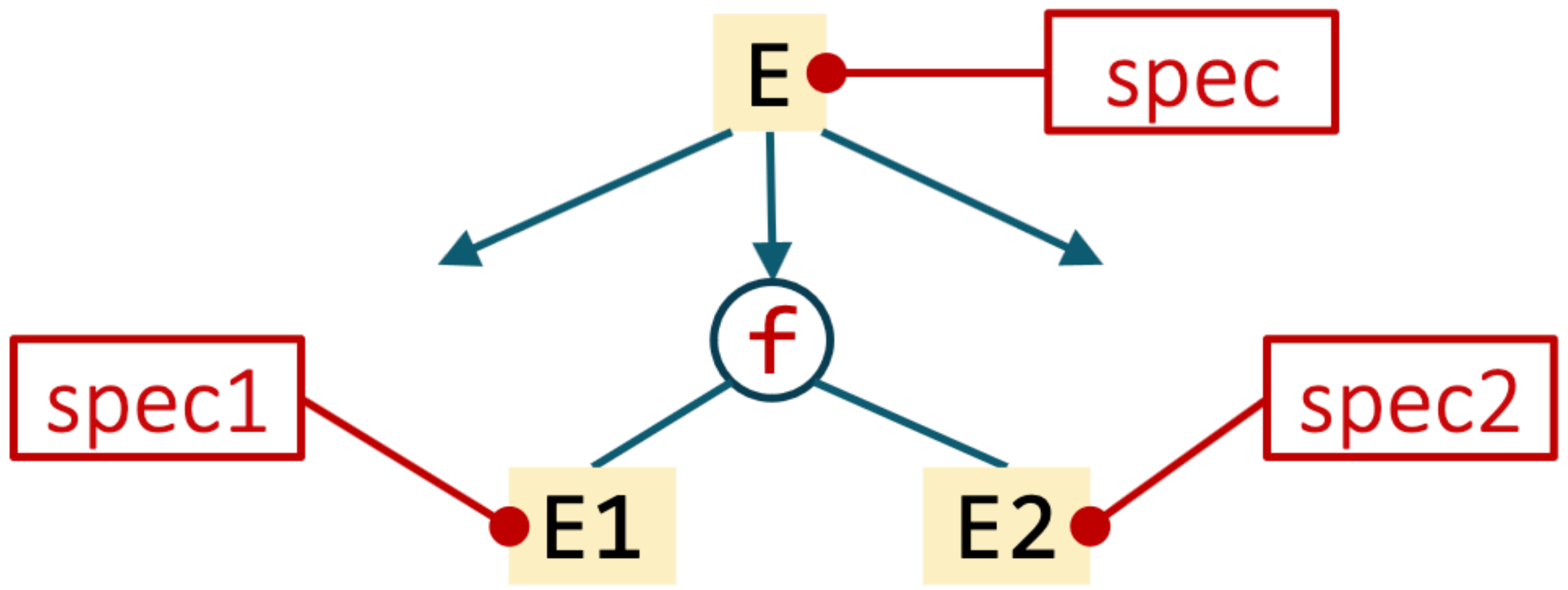
iter 0:	L			
iter 1:	x	L[N..N]		
iter 2:	L[N..N]			
iter 3:	x[N..N]	L[N..N][N..N]		
iter 4:	x[0..N]	x[find(L,N)..N]	L[N..N][N..N]	
iter 5:	x[0..0]	x[0.. find(L,N)]	x[find(L,N)..N]	…
iter 6:	x[0.. find(L,N)]	x[find(L,N)..N]	…	
iter 7:	x[0.. find(x,N)]	x[0.. find(L[N..N],N)]	…	
iter 8:	x[0.. find(x,0)]	x[0.. find(x,find(L,N))]	…	
iter 9:				

[[1,4,0,6] → [1,4]]

Need to reject useless programs early in the search!

Top-down propagation of the spec

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

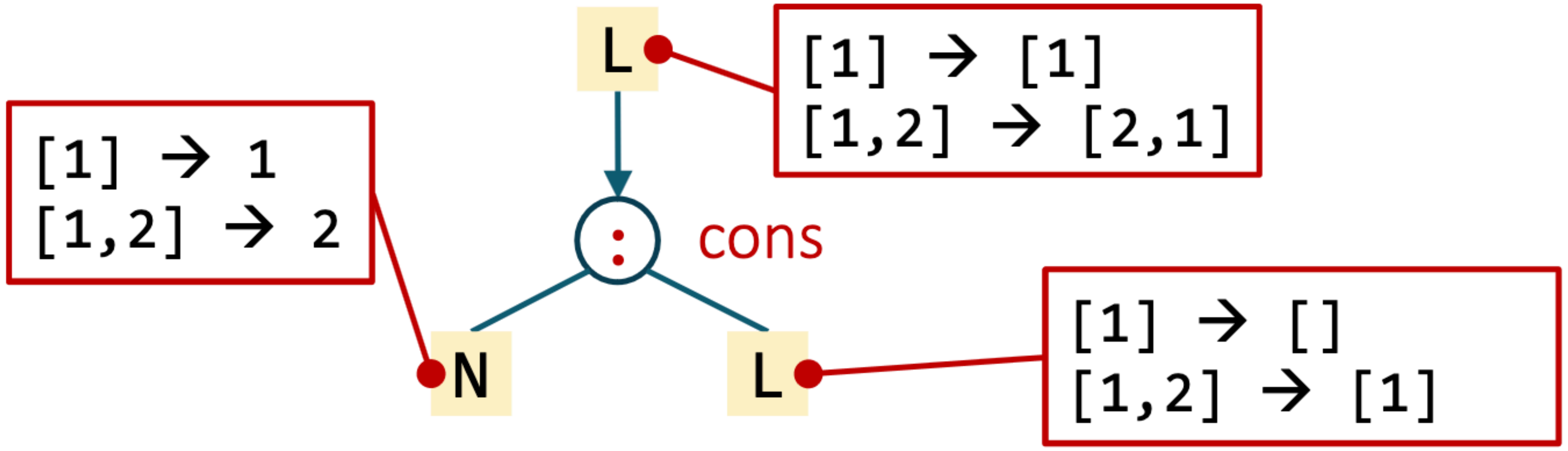


Now is spec1 = ⊥ or spec2 = ⊥ then discard the expansion of the set of terms of the form f (E1, E2).

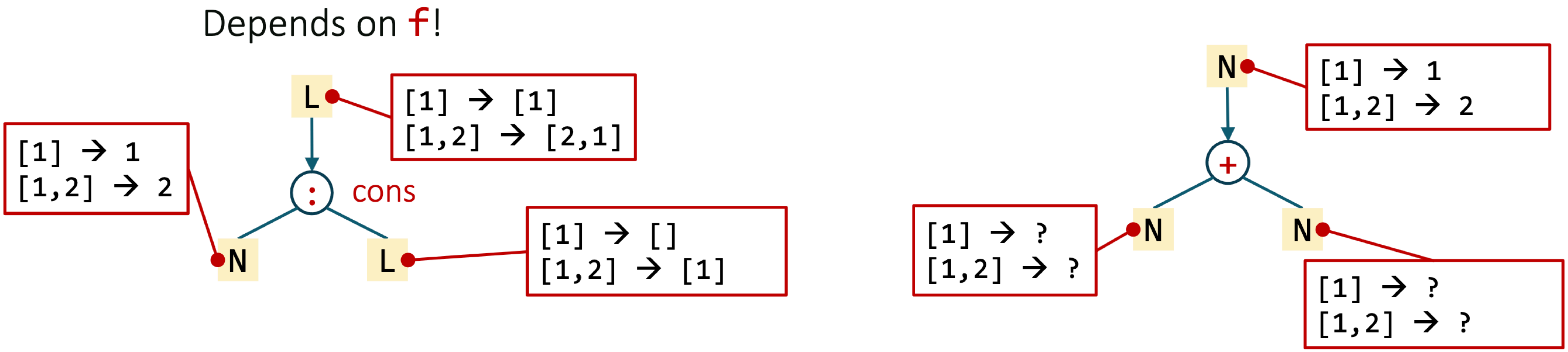
Currently : Spec = examples

When is TDP possible?

Depends on f!

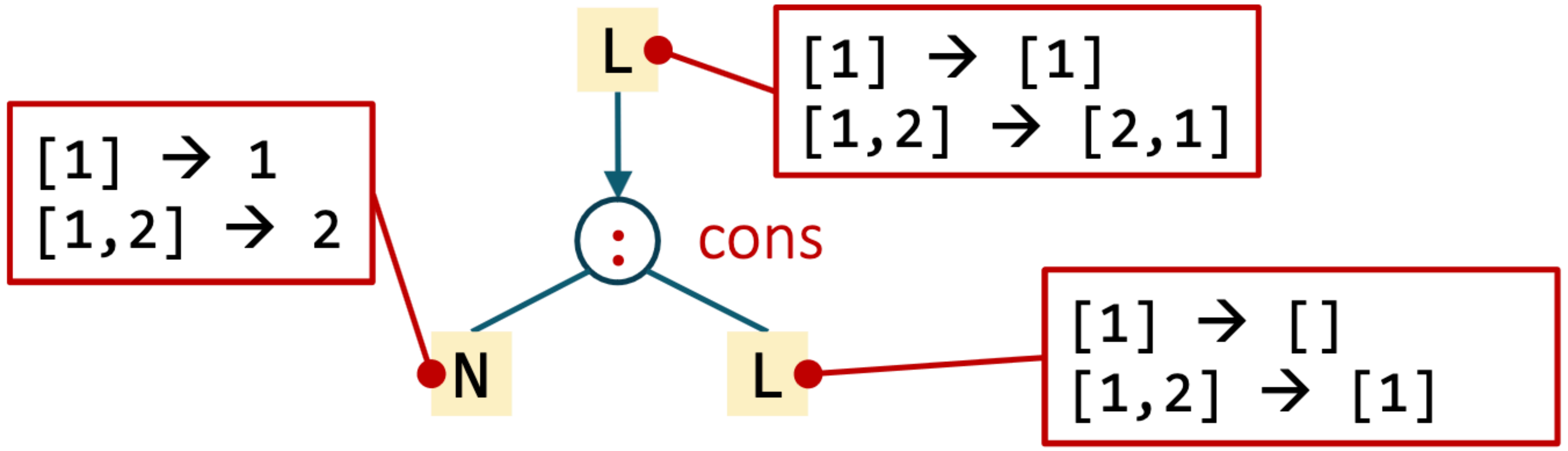


When is TDP possible?



When is TDP possible?

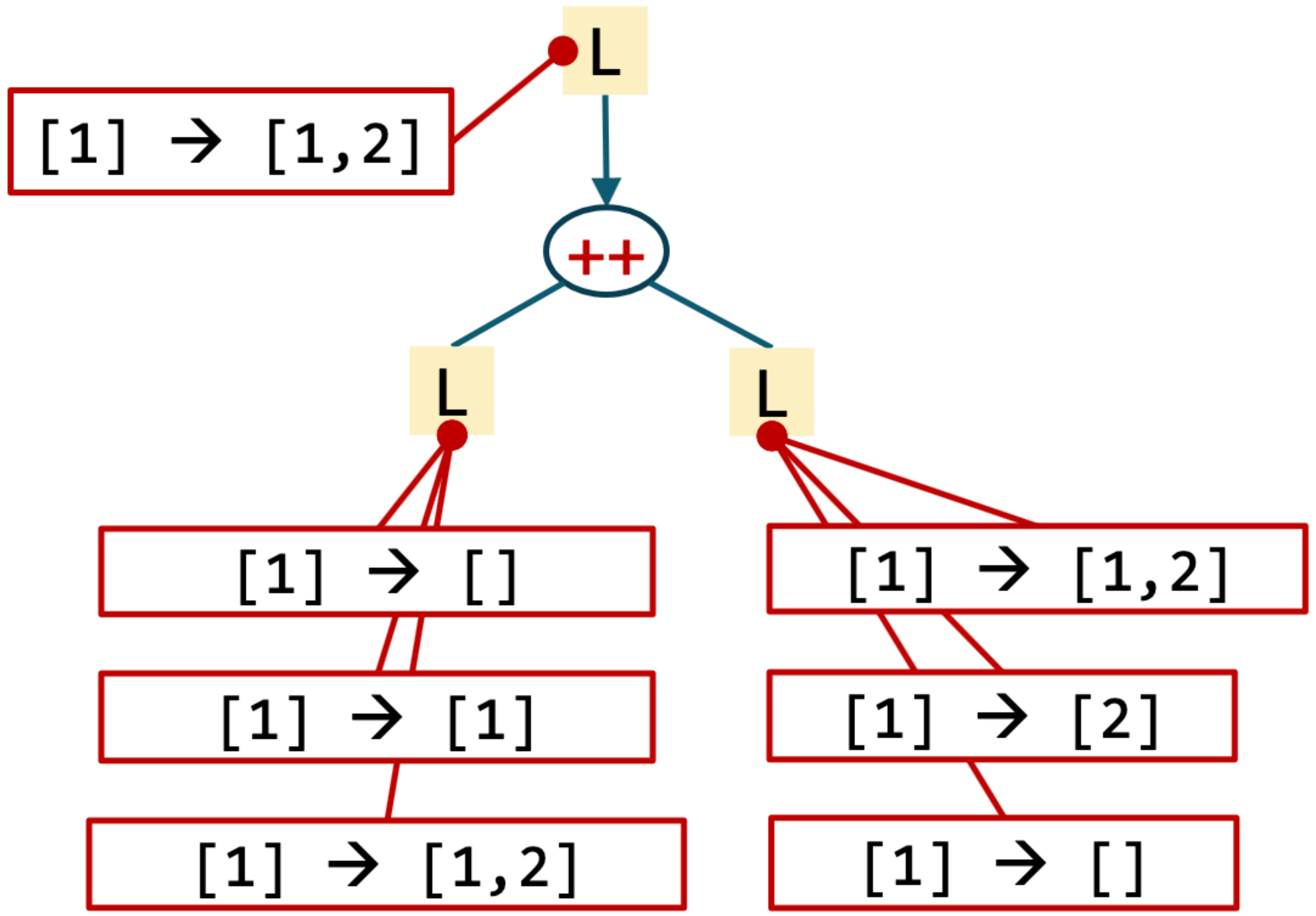
Depends on f!



The inverse semantics is uniquely defined

Works when the function is injective!

Something less strict



Works when the function has a "small inverse"

- or just the output examples have a small inverse

FlashFIll work uses this property for functions over spreadsheets.

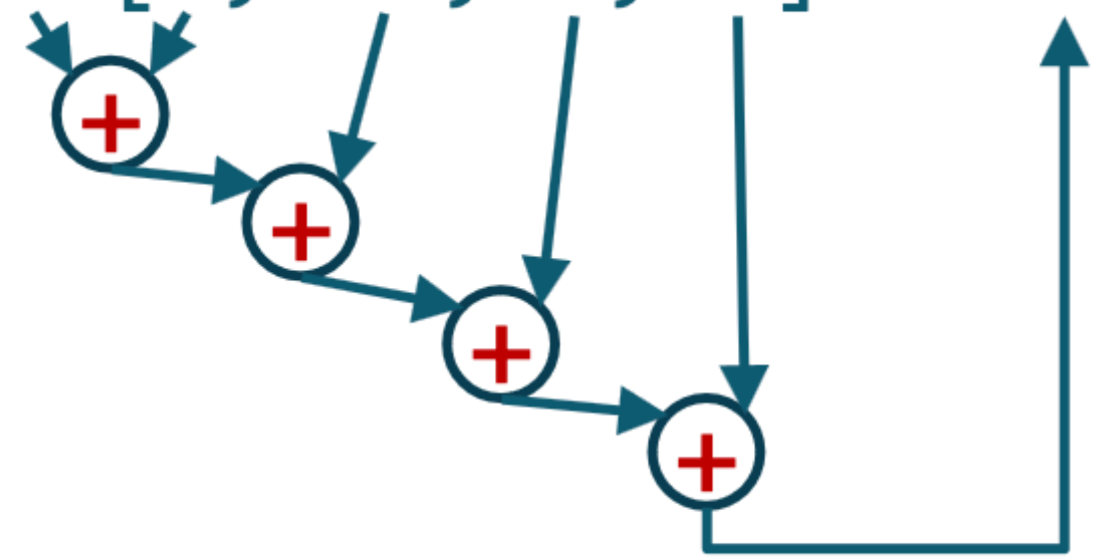
λ^2: TDP for list combinators

[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
```

Functional Idioms

map *f lst* = case *lst* of
    [] -> []
    head:*rest* -> *f*(*head*) : (map *f rest*)

- Applies *f* to every element in the list

filter *p lst* = case *lst* of
    [] -> []
    head:*rest* -> if p(*head*) then *head*: (filter *p rest*)
        else (filter *p rest*)

- Removes any element x for which p(x) is false

foldl definition

```
foldl binop start lst = case lst of
                  []  -> start
                   head:rest -> (foldl binop (binop start head) rest)
```

- Apply the binary operation binop from left to right to the list

foldl definition

```
foldl binop start lst = case lst of
                   []  -> start
                    head:rest -> (foldl binop (binop start head) rest)
```

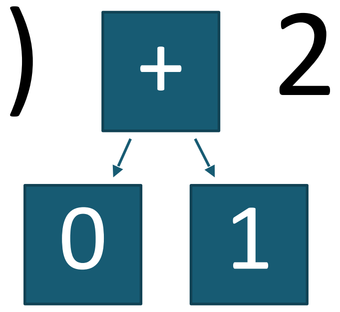
- Apply the binary operation binop from left to right to the list

```
foldl (+)  0  1:2:3:4:[]
```

foldl definition

```
foldl binop start lst = case lst of
                []  -> start
                head:rest -> (foldl binop (binop start head) rest)
```

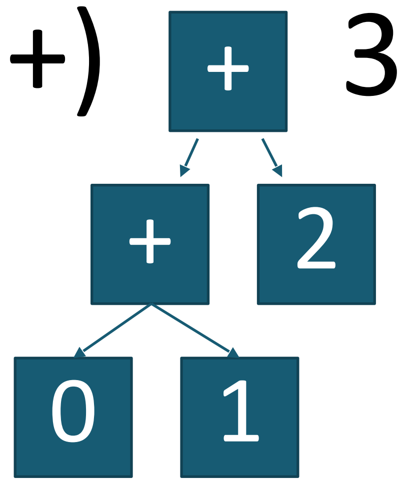
- Apply the binary operation binop from left to right to the list

foldl (+)  2:3:4:[]

foldl definition

```
foldl binop start lst = case lst of
                    []  -> start
                     head:rest -> (foldl binop (binop start head) rest)
```

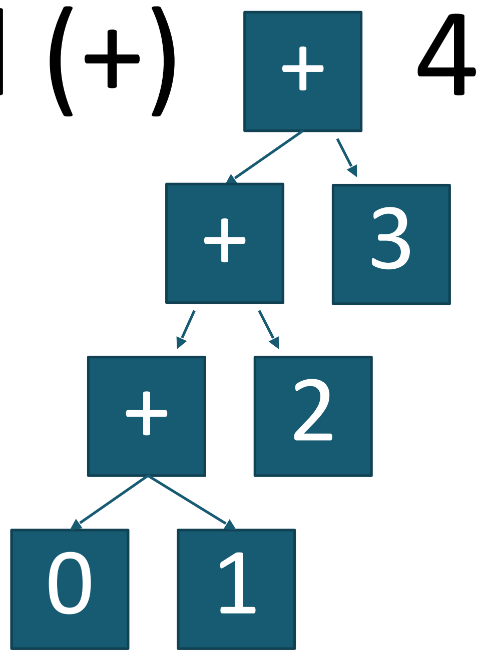
- Apply the binary operation binop from left to right to the list

foldl (+)  3:4:[]

foldl definition

```
foldl binop start lst = case lst of
                    []  -> start
                     head:rest -> (foldl binop (binop start head) rest)
```

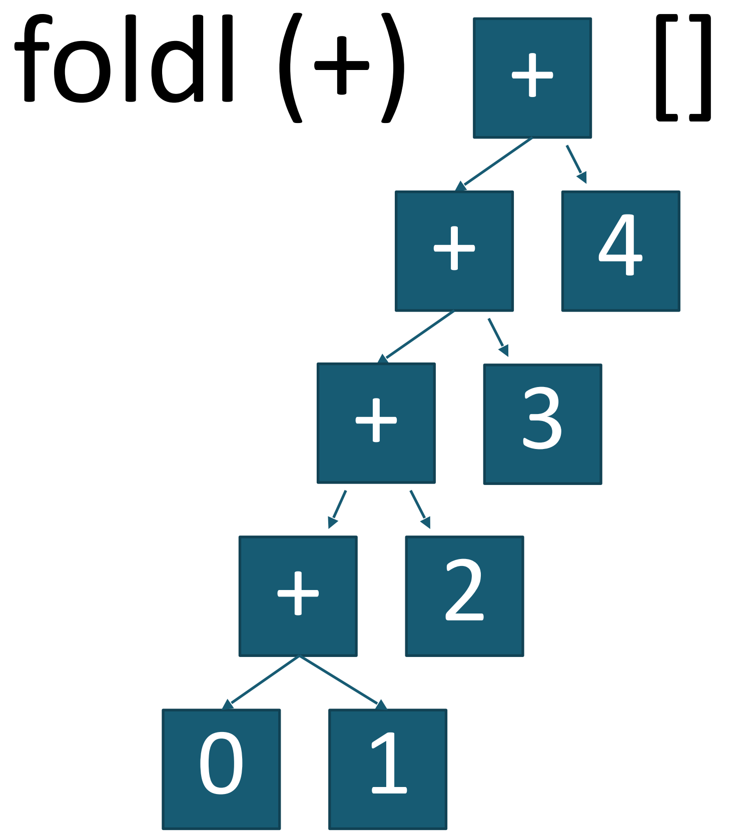
- Apply the binary operation binop from left to right to the list

foldl (+)  4:[]

foldl definition

```
foldl binop start lst = case lst of
                    []  -> start
                    head:rest -> (foldl binop (binop start head) rest)
```

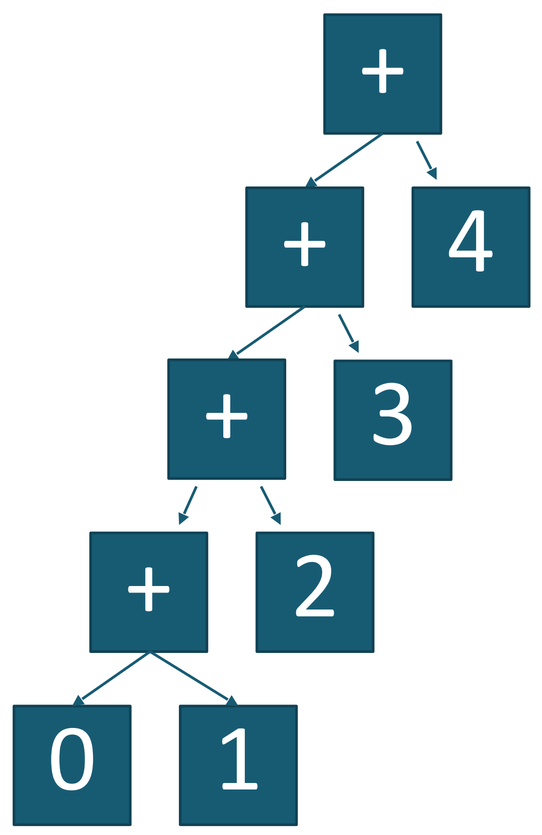
- Apply the binary operation binop from left to right to the list



foldl definition

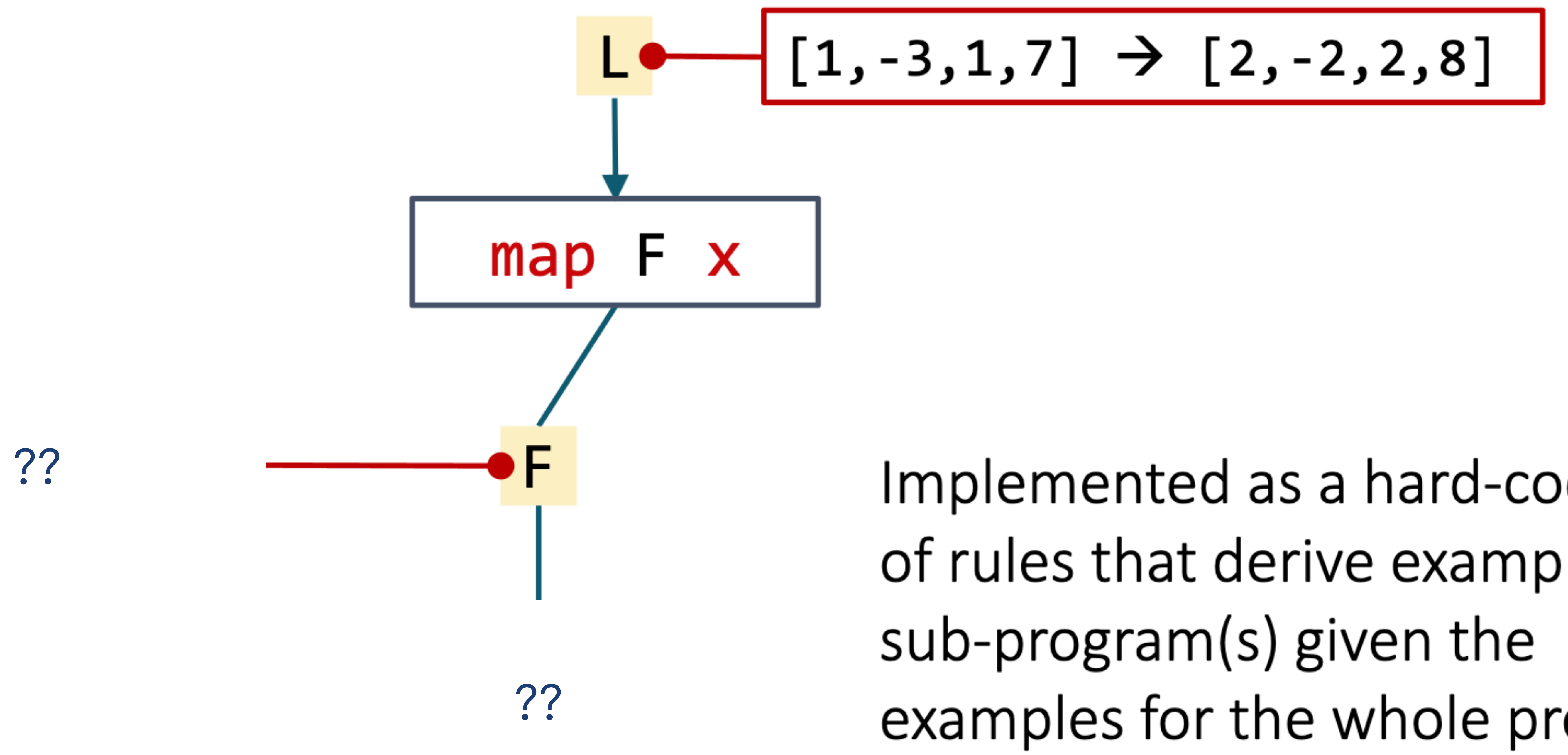
```
foldl binop start lst = case lst of
                    []  -> start
                     head:rest -> (foldl binop (binop start head) rest)
```

- Apply the binary operation binop from left to right to the list



λ^2: TDP for list combinators

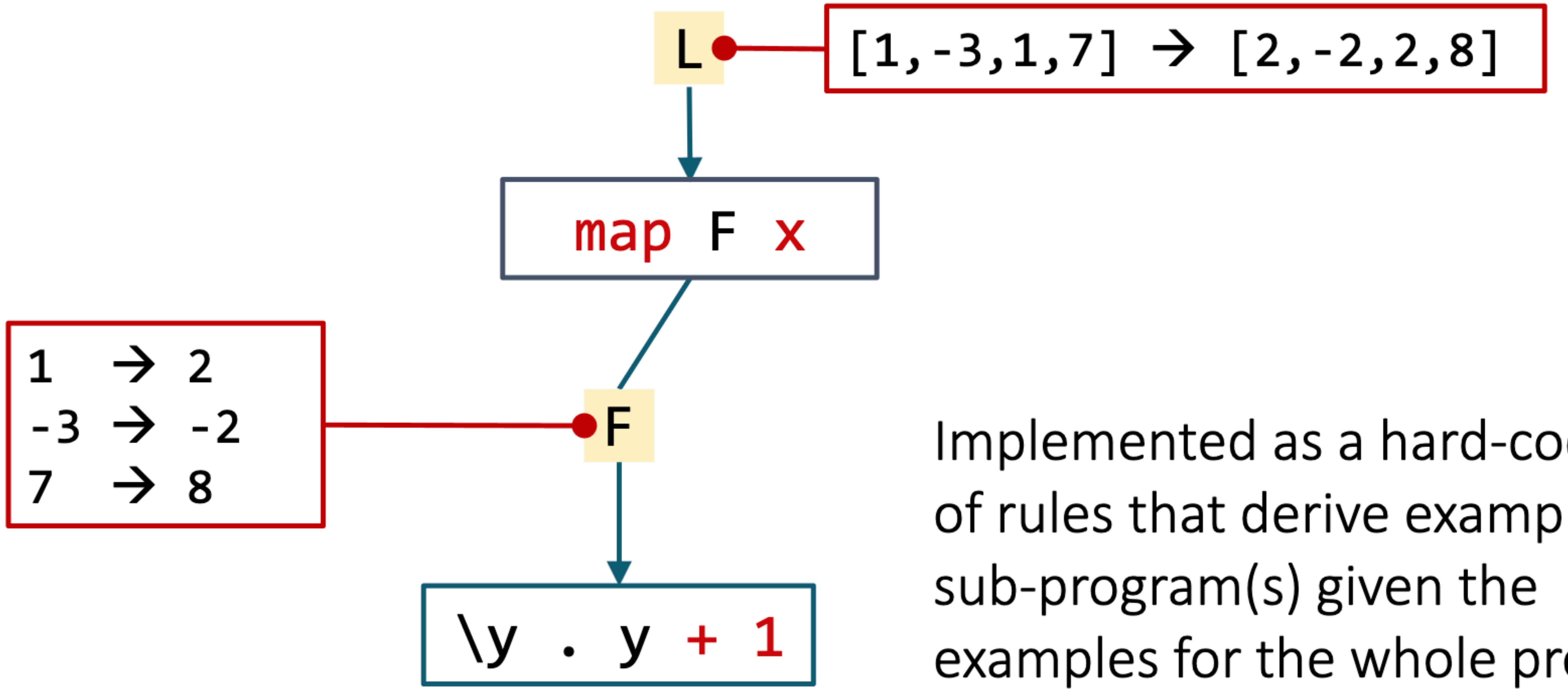
[Feser, Chaudhuri, Dillig '15]



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

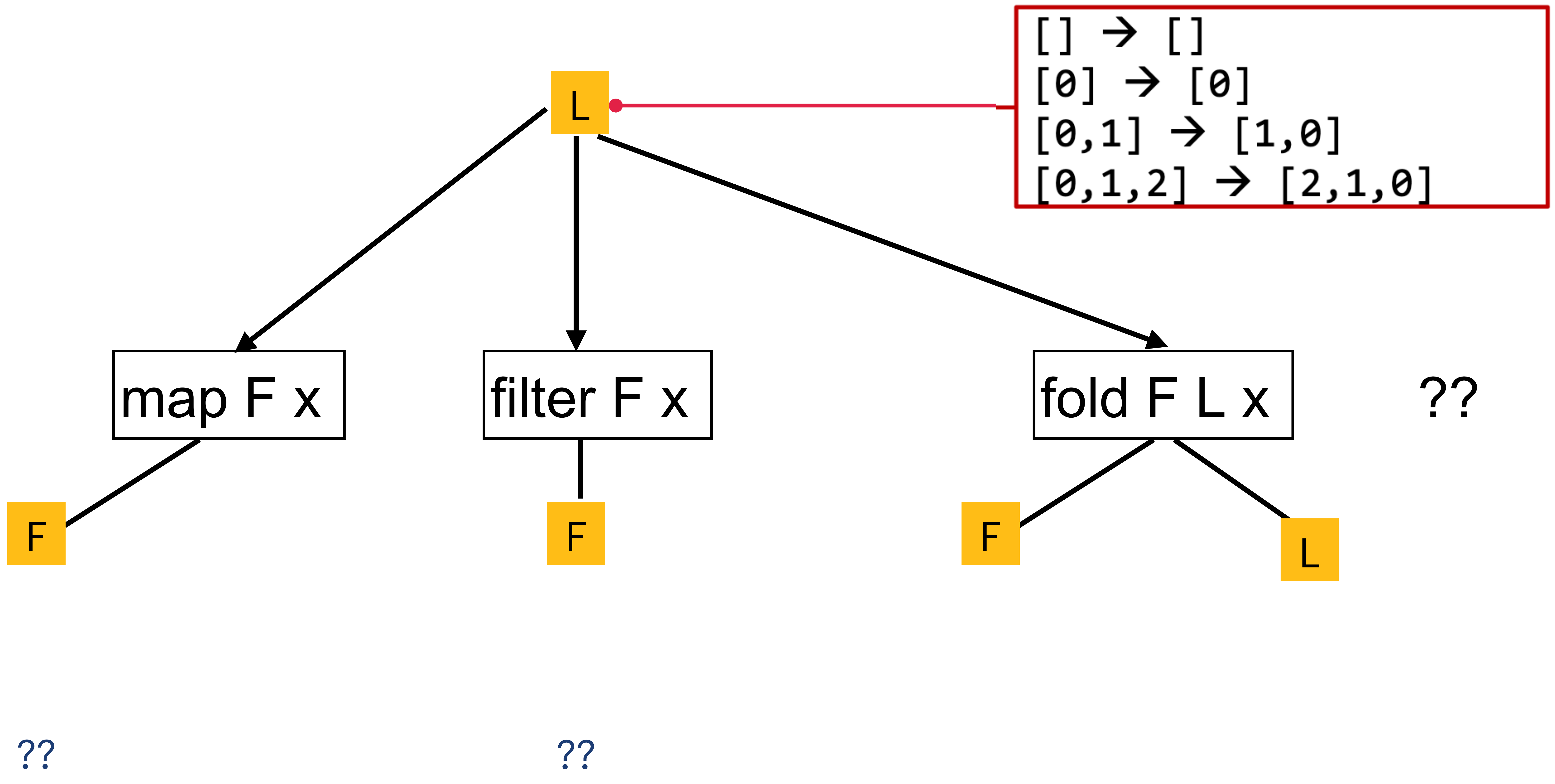
λ^2: TDP for list combinators

[Feser, Chaudhuri, Dillig '15]

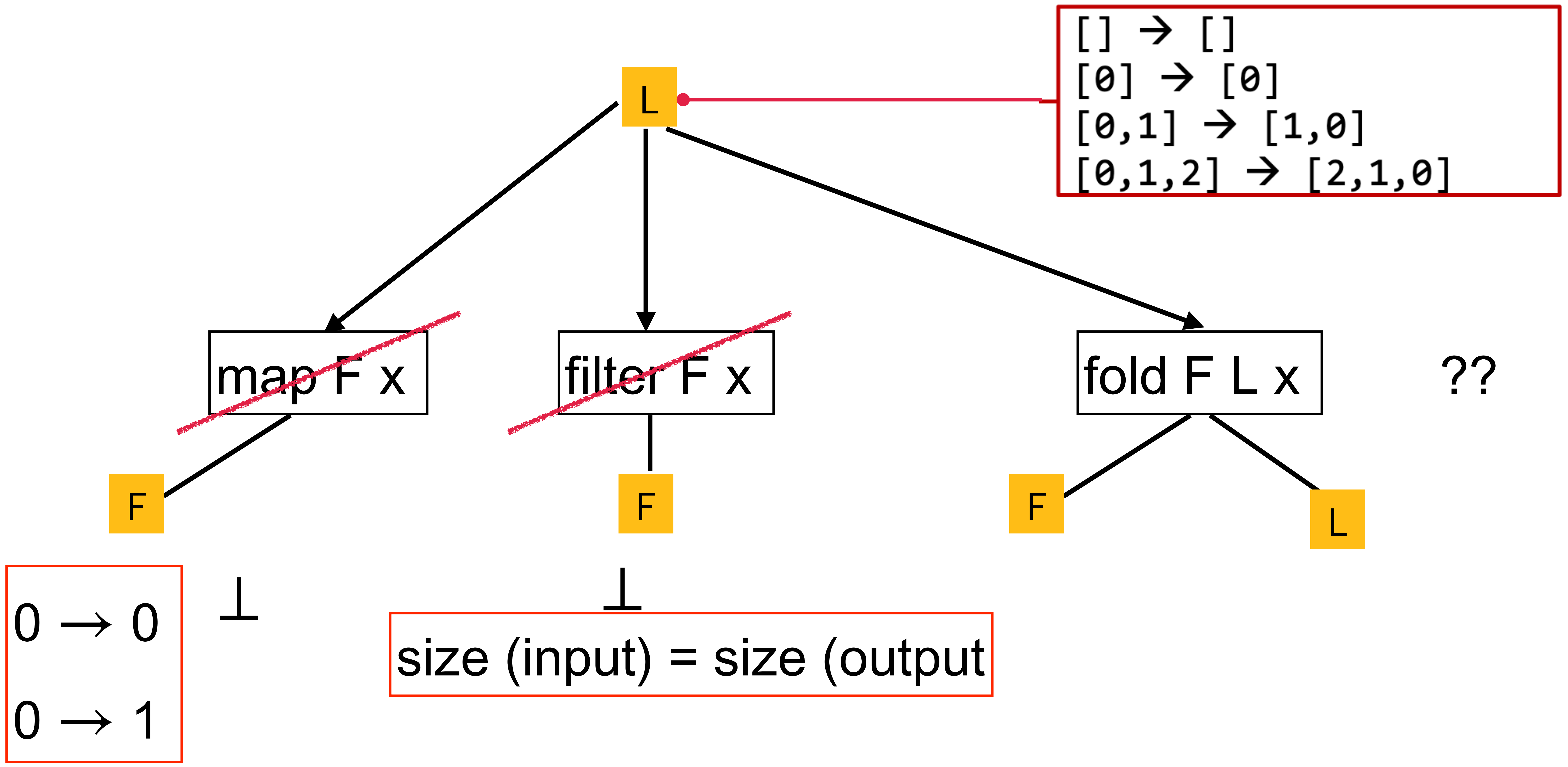


Implemented as a hard-coded set of rules that derive examples for sub-program(s) given the examples for the whole program and the combinator

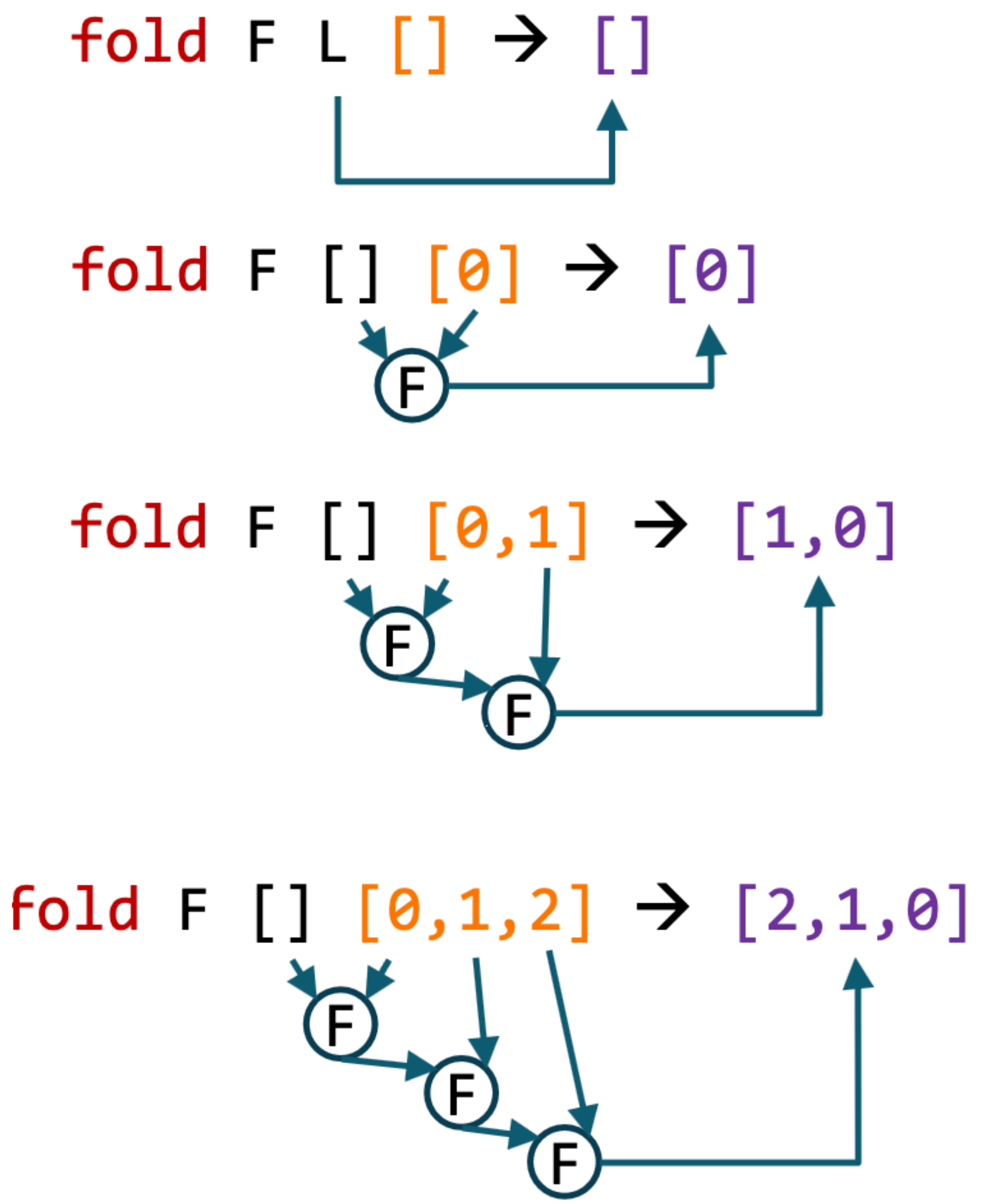
λ^2: TDP for list combinators

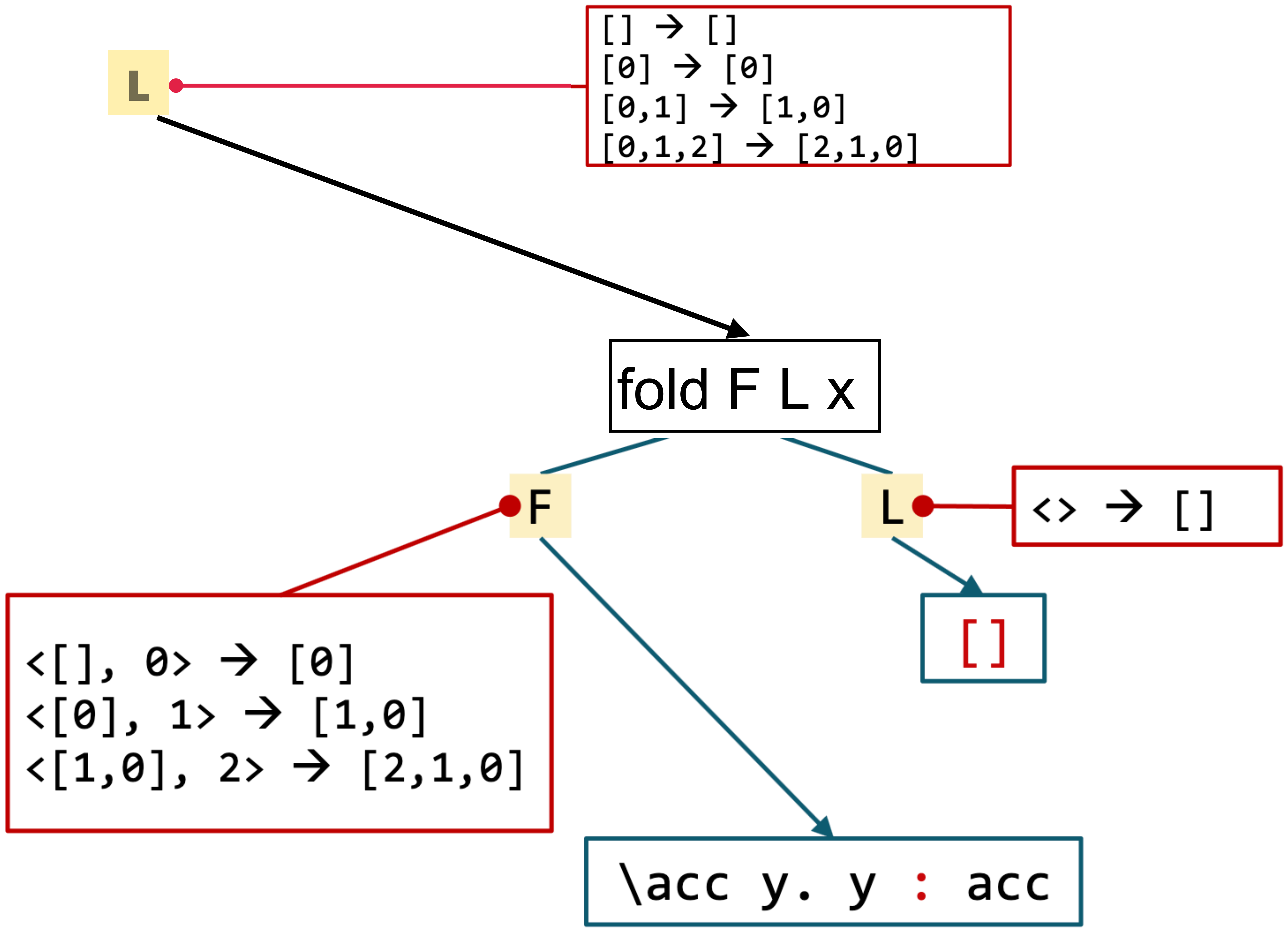


λ^2: TDP for list combinators



λ^2: TDP for list combinators





Condition abduction

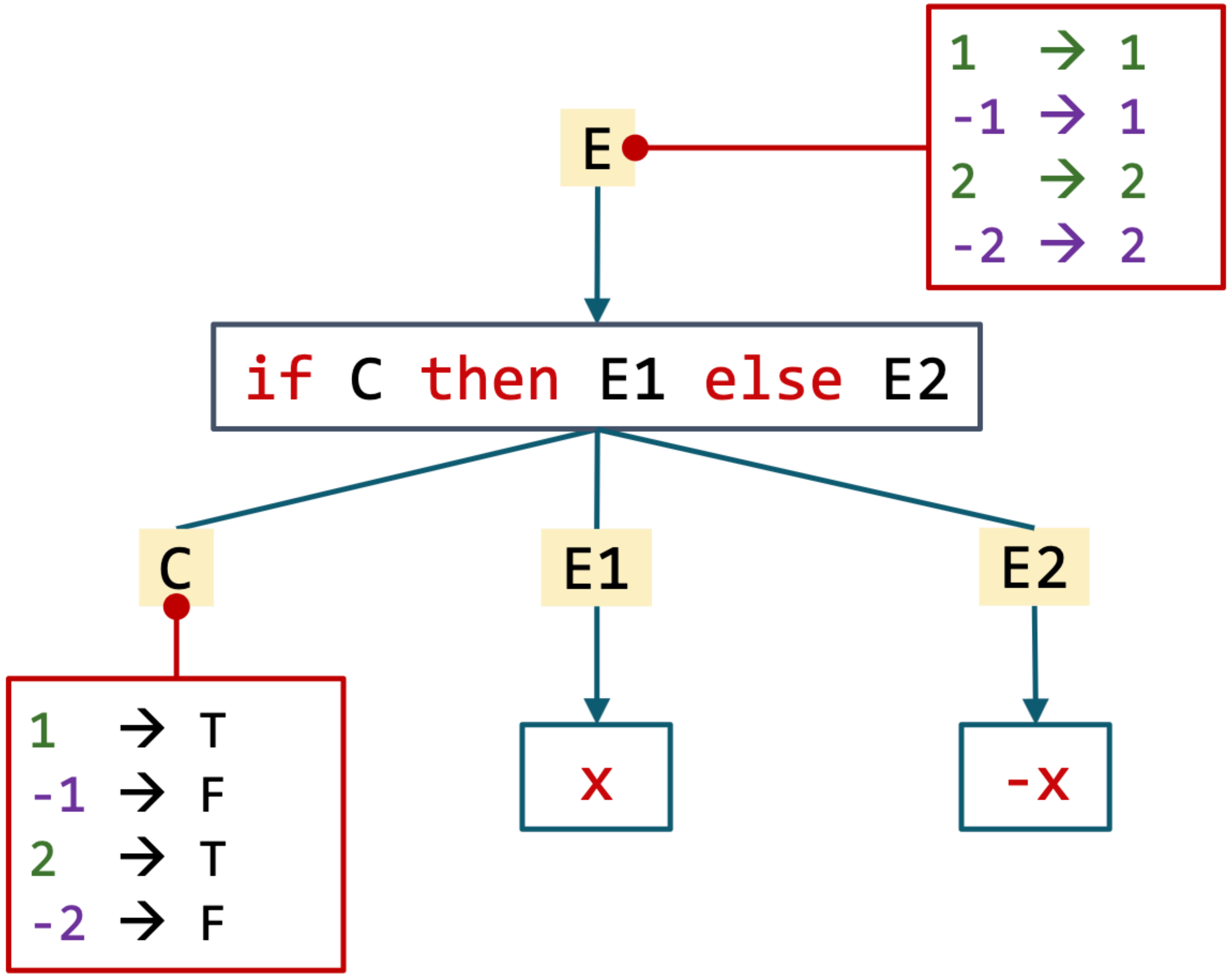
Smart way to synthesize conditionals

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



Types and Type based Top-down pruning

Example

Drop the smallest element from each list

$$\begin{bmatrix} [71, & 75, & 83] \\ [90, & 87, & 95] \\ [68, & 77, & 80] \end{bmatrix} \Rightarrow \begin{bmatrix} [75, & 83] \\ [90 & 95] \\ [77 & 80] \end{bmatrix}$$

Example

$$\begin{bmatrix} [71, & 75, & 83] \\ [90, & 87, & 95] \\ [68, & 77, & 80] \end{bmatrix} \Rightarrow \begin{bmatrix} [75, & 83] \\ [90 & 95] \\ [77 & 80] \end{bmatrix}$$

How can we discover this program?

```
dropmins x = map dropmin x
    where dropmin y = filter isNotMin y
        where isNotMin z = foldl h False y
            where h t w = t || (w < z)
```

Defining the language

expr = var

| $\lambda x\,.\ expr$

| **filter** *expr expr*

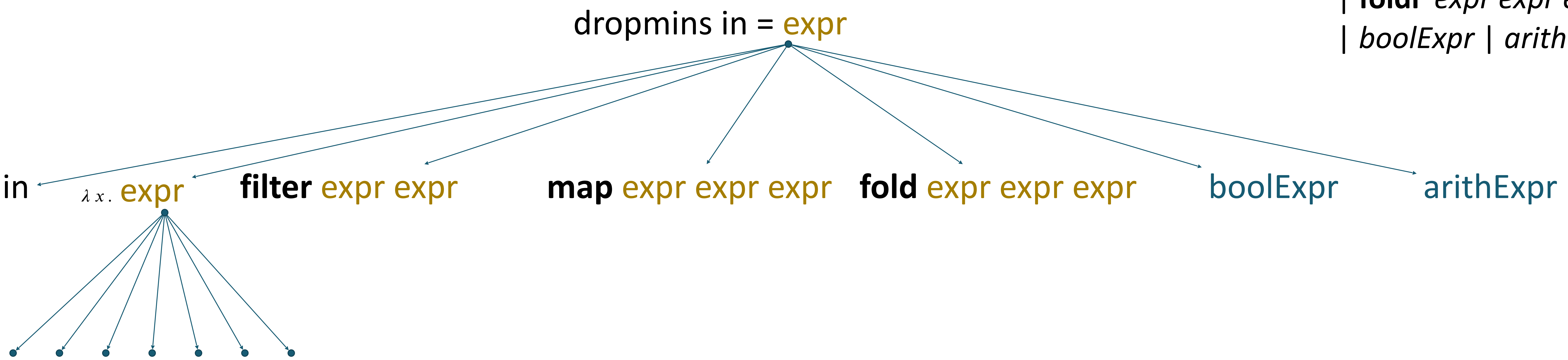
| **map** *expr expr*

| **foldl** *expr expr expr*

| *boolExpr* | *arithExpr*

Top-down search

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



Many of these programs can be eliminated before having to complete them!

How?

Top-down search

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*

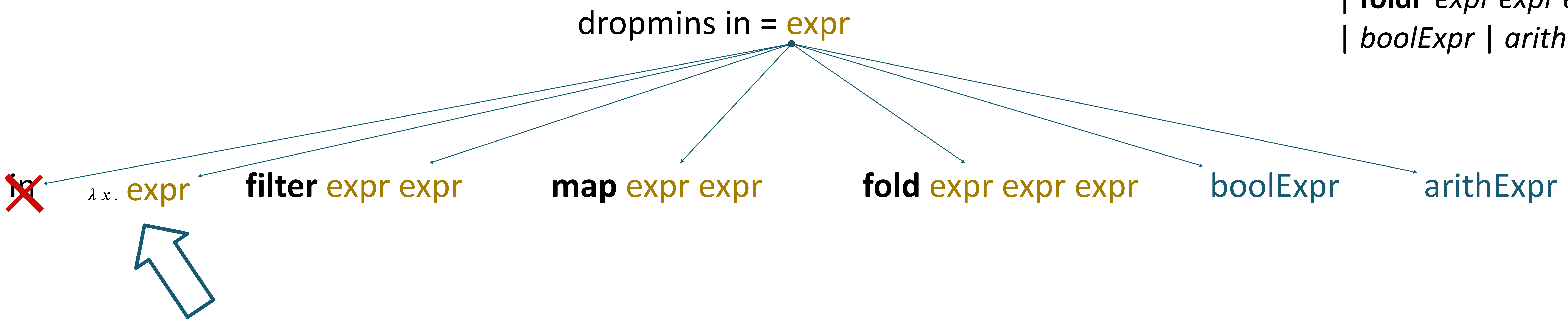
dropmins in = expr

in | $\lambda x.$ expr | **filter** expr expr | **map** expr expr | **fold** expr expr expr | boolExpr | arithExpr

This is a fully concrete program, and it clearly doesn’t match the examples

Top-down search

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



This program has a missing expression, but we can already tell it will not work. Why not?

Types

Our simple language supports an infinite set of types of 3 basic kinds

$$\tau \;:=\; Int \;\mid\; Bool \;\mid\; [\tau] \;\mid\; \tau \rightarrow \tau$$

Integer

Boolean

List of some type

Function from some type to some other type

Types

$$\begin{bmatrix} [71, & 75, & 83] \\ [90, & 87, & 95] \\ [68, & 77, & 80] \end{bmatrix}$$

$$[\ [Int]\ ]$$

$$\begin{bmatrix} [75, & 83] \\ [90 & 95] \\ [77 & 80] \end{bmatrix}$$

$$[\ [Int]\ ]$$

Input and output types are lists of lists of integers

Types

Each element in our language has a type given by a *typing rule*

$$\frac{premises}{C \vdash expr : \tau}$$

A typing rule like the one above states that $expr$ has type τ in a context C as long as all the premises are satisfied

- A context simply tracks information about the type of any variables

Types

Each element in our language has a type given by a *typing rule*

$$\frac{C \text{ says var} \quad \textit{has type } \tau}{C \vdash \text{var} : \tau}$$

$$\frac{f : \tau_1 \rightarrow \tau_2 \qquad epxr : \tau_1}{C \vdash f\ expr : \tau_2}$$

$$\frac{C, x : \tau_1 \vdash expr : \tau_2}{C \vdash \lambda\, \text{x}\,.\ \text{expr} : \tau_1 \rightarrow \tau_2}$$

$$\frac{}{map : (\tau_1 \rightarrow \tau_2) \rightarrow [\tau_1] \rightarrow [\tau_2]}$$

$$\frac{}{foldl : (\tau_{start} \rightarrow \tau_{lst} \rightarrow \tau_{start}) \rightarrow \tau_{start} \rightarrow [\tau_{lst}] \rightarrow \tau_{start}}$$

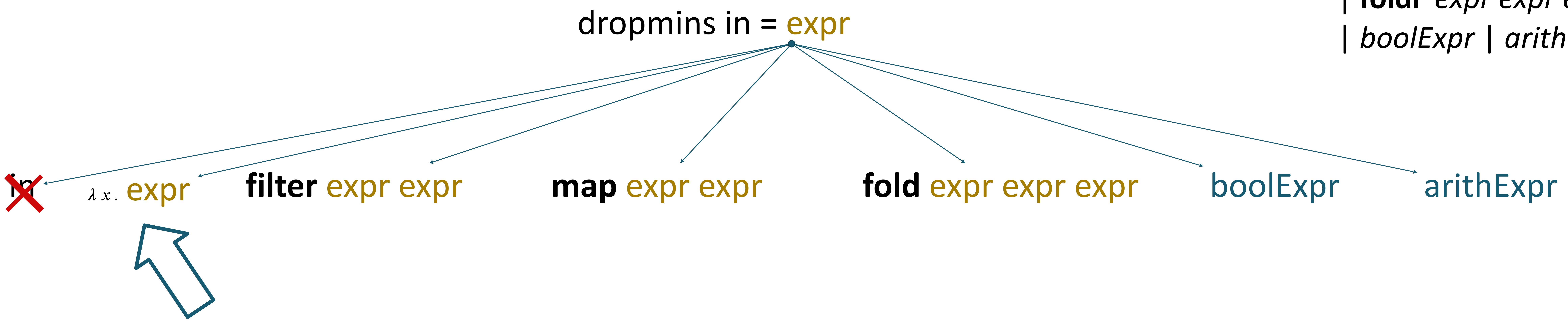
$$\frac{}{boolExpr : Bool}$$

$$\frac{}{filter : (\tau \rightarrow Bool) \rightarrow [\tau] \rightarrow [\tau]}$$

$$\frac{}{intExpr : Int}$$

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



$$\frac{expr : \tau_2 \text{ } assuming \text{ } x : \tau_1}{\lambda\, x.\ \text{expr} : \tau_1 \rightarrow \tau_2}$$

Based on the rule, this expression will have a type $\tau_1 \rightarrow \tau_2$

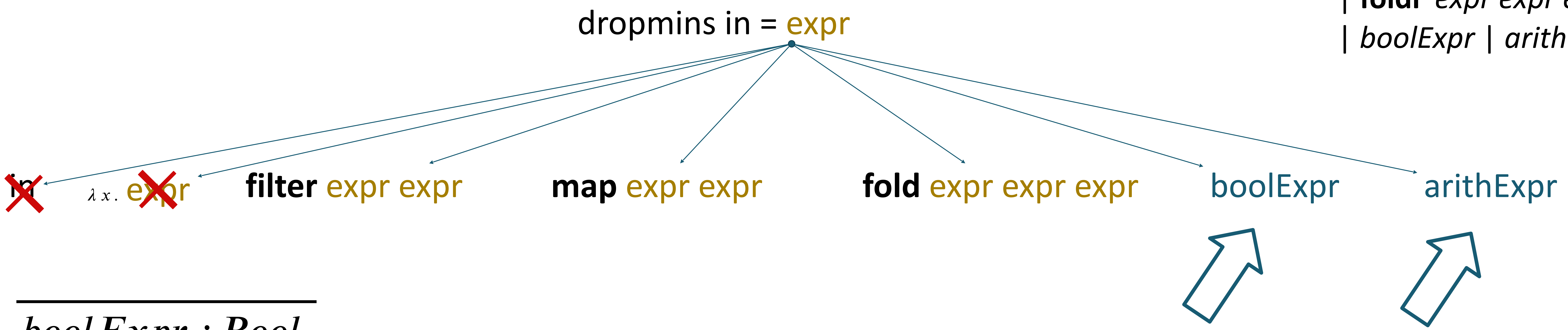
But we know the output must have type $[\ [Int]\ ]$

There is no way those types can be made equal,

so we can discard this expression!

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



$$\frac{}{boolExpr : Bool}$$

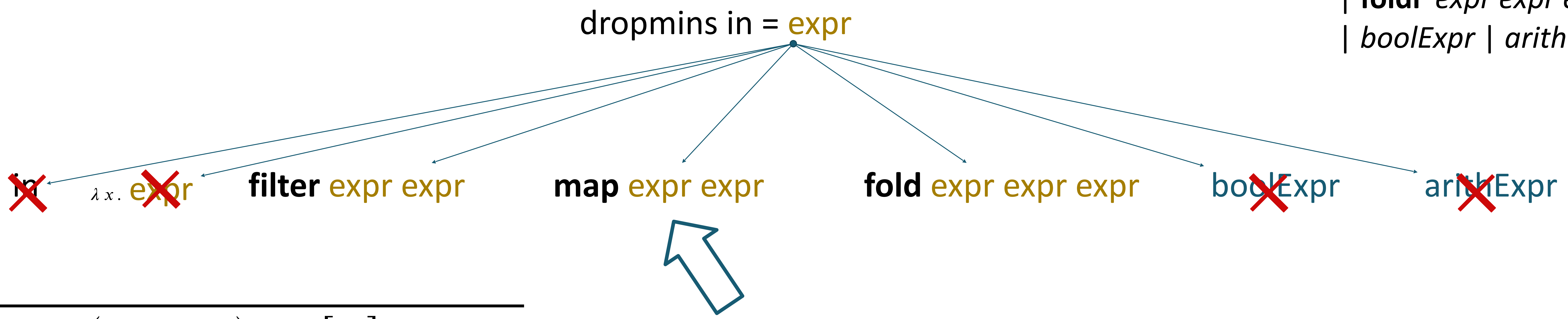
$$\frac{}{intExpr : Int}$$

With the same reasoning we can discard both of these expressions

They cannot possibly have the correct type

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



$$map: (\tau_1 \to \tau_2) \to [\tau_1] \to [\tau_2]$$

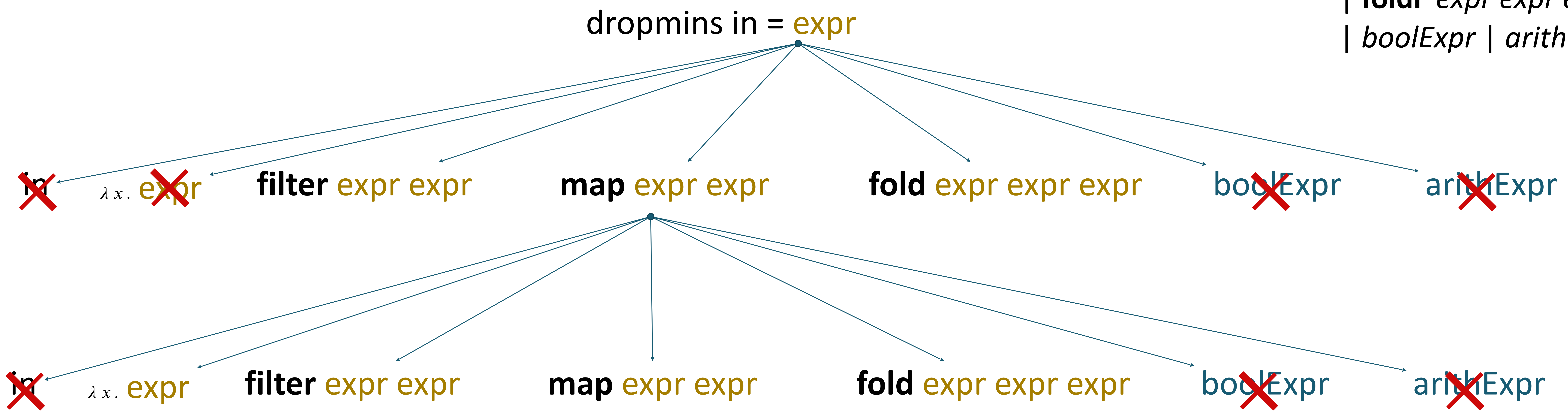
We know the output should be $[[Int]]$

This means the first expr must be $\tau_1 \to [Int]$

otherwise the types won't match

Type-based pruning

expr = var
| $\lambda x.\ expr$
| **filter** *expr expr*
| **map** *expr expr*
| **foldl** *expr expr expr*
| *boolExpr* | *arithExpr*



We can quickly dismiss many possible expressions because they cannot produce the type $\tau_1 \to [Int]$

The Paper

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?
  - How would it break the enumerative solver?

EUSolver

- Q2: What are pruning/decomposition techniques EUSolver used 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?
- Can we discard a term that covers a subset of the points covered by another term?

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

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

No solution size evaluation beyond those solved by ESolver

No ablation of DT repair / branch-wise verification

Counterexample-Guided Quantifier Instantiation for Synthesis in SMT, CAV ’15

Next Week.

- Review of logic:
  - Propositional and FO logic.
  - Satisfiability and Validity of Logical Formulas.
- SAT solvers.
- SMT solvers.

- I will assign a reading for this by tomorrow!

End