CS5733 Program Synthesis

#3.Optimizing the Enumerative Search

Ashish Mishra

Logistics

- Reviews:
  - Due tomorrow
- Projects:
  - Topics: due next Thursday
- Other questions?

Academic paper reading workshop… on steroids

- Abstract and Introduction. Sec. 0 and 1.
- Overview:
  - Understand the Problem and the Solution. Sec. 2 and 3.
- Technical Discussions: Sec. 4 - ?X (Anything Before Evaluation)
  - Dig deeper into the Solution
  - and/or formally understand the problem.
- Proofs/Evaluation/Implementation: Sec. ?X
  - Proofs for theorems — Typically in PL Papers —-
  - Evaluations of the ideas explained in Sec 2&3 and Discussed in Sec 4-?X
    - Comparison with an appropriate baseline.
  - Sometimes other specific RQs
- Conclusion, Related work, Future works.

EUSolver as an Example

SyGuS Continue...

Enumerative Search

=

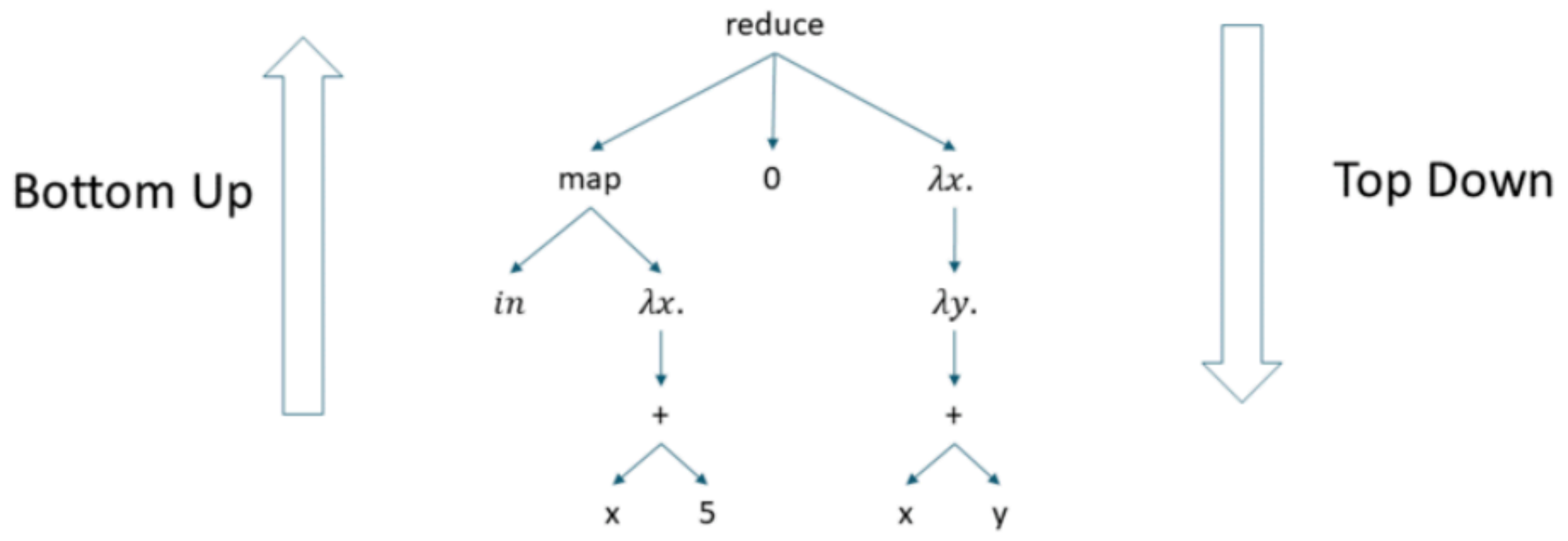
Explicit / Exhaustive Search

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

Challenge: How do we systematically enumerate all programs?

top-down vs bottom-up

reduce (map in λ x. x + 5) 0 $\lambda x. \lambda y. x + y$



FP Trivia:

reduce (map in $\lambda x \,.\, x + 5$) 0 $(\lambda x \,.\, \lambda y \,.\, (x + y))$

functions : map, reduce

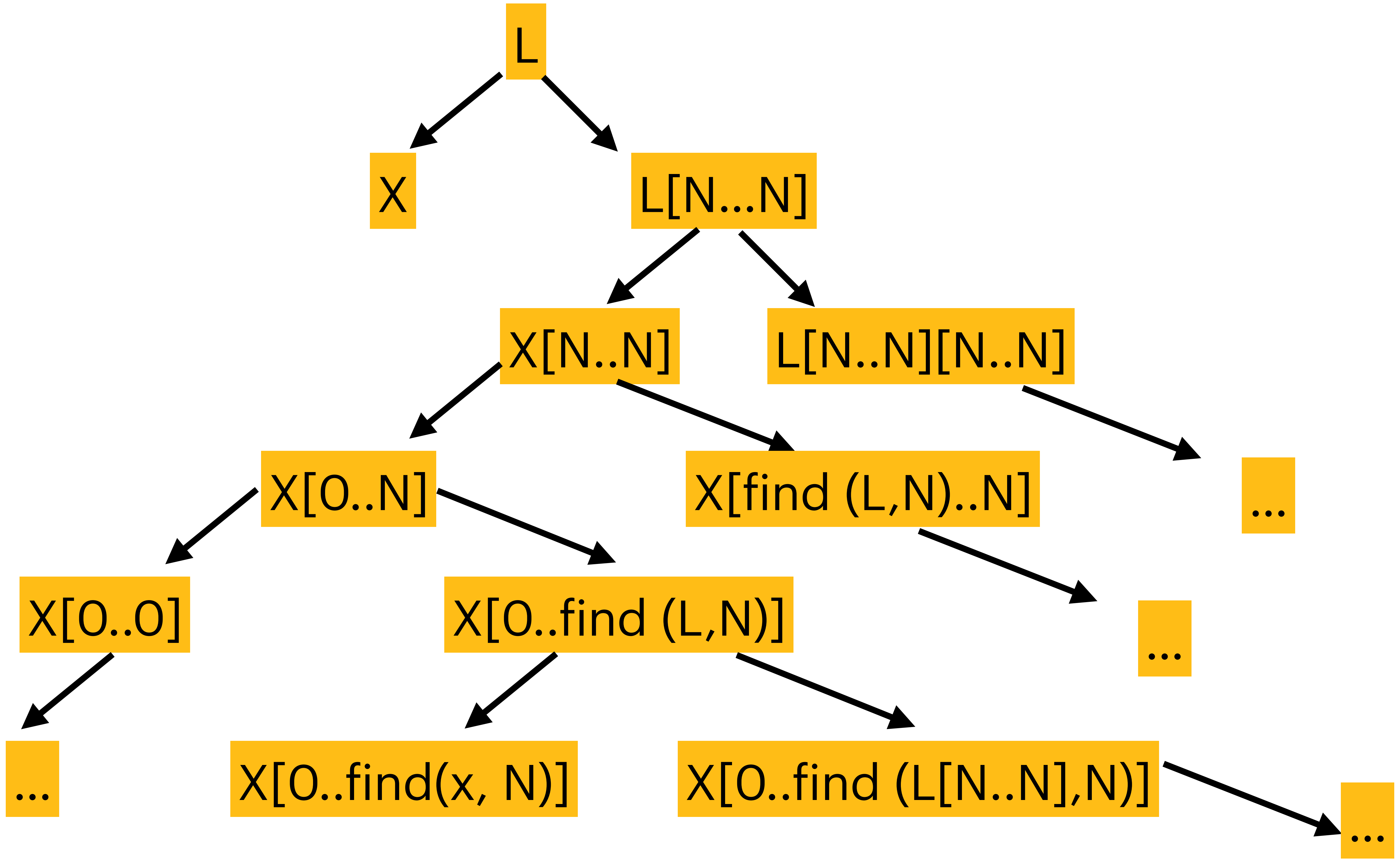
Top-down enumeration: search space

Search space is a tree where

- nodes are whole incomplete programs
- edges are “derives in one step”

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

[[1,4,0,6] → [1,4]]
```



L

X

L[N...N]

X[N..N]

L[N..N][N..N]

X[0..N]

X[find (L,N)..N]

...

X[0..0]

X[0..find (L,N)]

...

...

X[0..find(x, N)]

X[0..find (L[N..N],N)]

...

Top-down enumeration = traversing the tree

- Search tree can be traversed:
  - depth-first (for fixed max depth)
  - breadth-first
  - later in class: best-first
- General algorithm:
  - Maintain a worklist of incomplete programs
  - Initialize with the start non-terminal
  - Expand left-most non-terminal using all productions

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

[[1,4,0,6] → [1,4]]
```

Top-down Algorithm

nonterminals rules (productions)

alphabet starting nonterminal

```
top-down(<Σ, N, R, S>, [i → o]):
  wl := [S]
  while (wl != []):
    τ:= wl.dequeue()
    if (complete(τ) ∧ τ([i]) = [o]):
      return τ
    wl.enqueue(unroll(τ))

unroll(τ):
  wl' := []
  A := left-most non-term in τ
  forall (A → rhs) in R:
    τ' = τ[A -> rhs]
    if !exceeds_bound(τ'): wl' += τ'
  return wl'
```

depth- or breadth-first
depending on where you enqueue

can impose bounds on depth/size

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

[[1,4,0,6]  →  [1,4]]
```

Top-down: example (depth-first)

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]]`

Bottom-up enumeration

The dynamic programming approach:

- Maintain a bank of complete programs
- Combine programs in the bank into larger programs using productions

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

```
[[1,4,0,6]  →  [1,4]]
```

Bottom-up: algorithm (take 1)

nonterminals
rules (productions)
alphabet
starting nonterminal

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank := {}
  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
        bank += t;

new-terms(A → σ($A_1 \ldots A_k$), d, bank):
 if (d = 0 ∧ k = 0) yield σ
 else forall <$t_1, \ldots, t_k$> in bank$^k$:
         if $A_i$ ->* $t_i$: yield σ($t_1, \ldots, t_k$)
```

inefficient, better index bank by non-terminal!

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

[[1,4,0,6] → [1,4]]

Bottom-up: algorithm (take 2)

nonterminals rules (productions)

alphabet starting nonterminal

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank := {}
  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
        bank += t;
```

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

[[1,4,0,6] → [1,4]]

```
new-terms($A \to \sigma(A_1 \ldots A_k)$, d, bank):
if (d = 0 ∧ k = 0)  yield $\sigma$
else forall $< t_1 \ldots t_k >$ in bank[$A_1$] $\times \ldots$ bank[$A_k$]]:
  yield σ(t 1,…,t k)
```

inefficient, generating same terms again and again!
better index bank by depth

Bottom-up: algorithm (take 3)

nonterminals rules (productions)

alphabet starting nonterminal

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank := {}
  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
        bank += t;
```

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

```
[[1,4,0,6] → [1,4]]
```

```
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($t_1$,…,$t_k$)
```

Bottom-up: example

Program bank

d= 0: `x` `0`

d =1: `sort(x)` `x + x` `x[0..0]` `[0]` `find(x,0)`

d = 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)` `sort([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)` `x[0..find(x,0)]` ✓

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

`[[1,4,0,6] → [1,4]]`

Explicit search from grammars

- Limitations:
  - Only scales to very small programs
  - Unsuitable for programs with unknown constants
    - A single unknown 32-bit constant makes the problem intractable
  - Hard to deal with context dependent semantics
- Example system:
  - Recursive Program Synthesis [Albarghouthi et al., CAV 2013]

Enumerative search

```
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)

Candidates are whole but might not be complete

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

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 subprograms

`m * N`^{2} `m * (N - 1)`^{2}

Prioritize

Explore more promising candidates first

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

← dequeue this first

Useless depends on the problem and the domain.

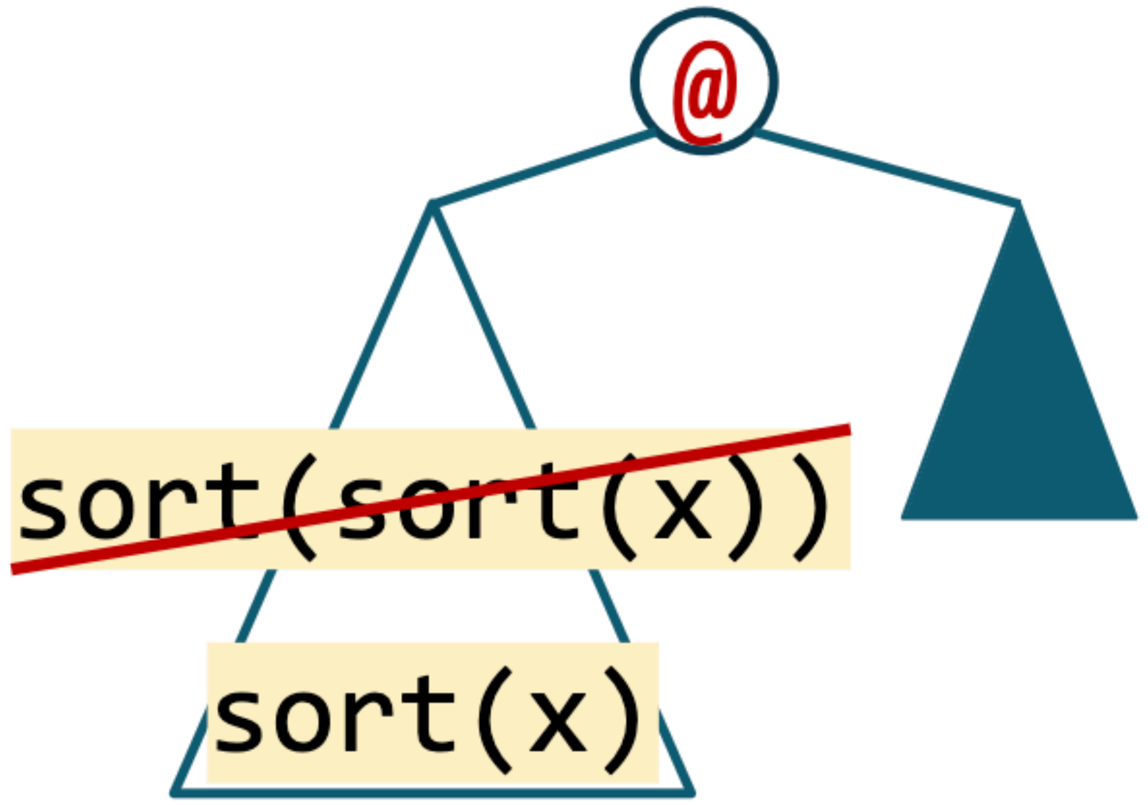
Equivalent terms

Terms guaranteed not to lead to a solution

Now this : Optimizing the Search

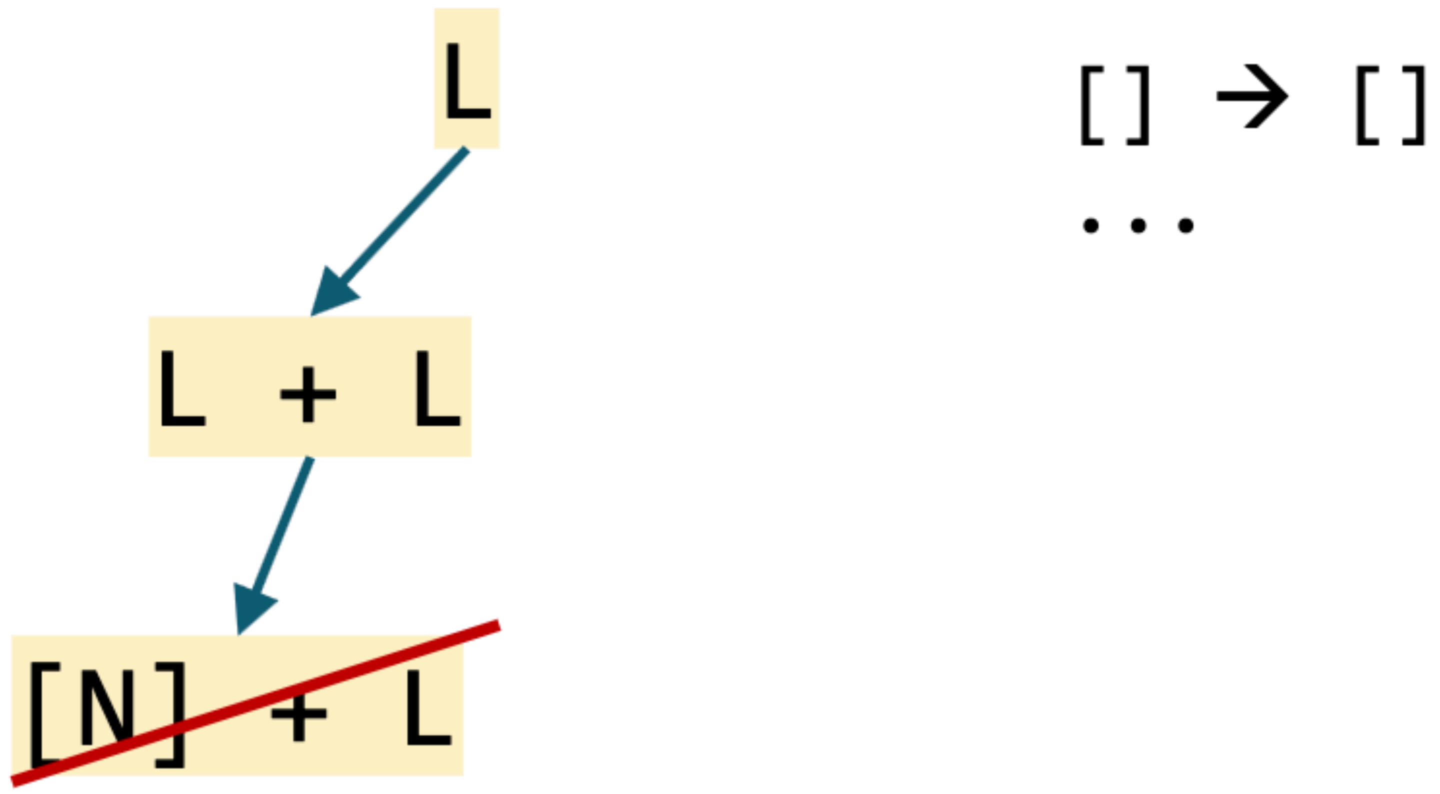
When can we discard a program?

redundant



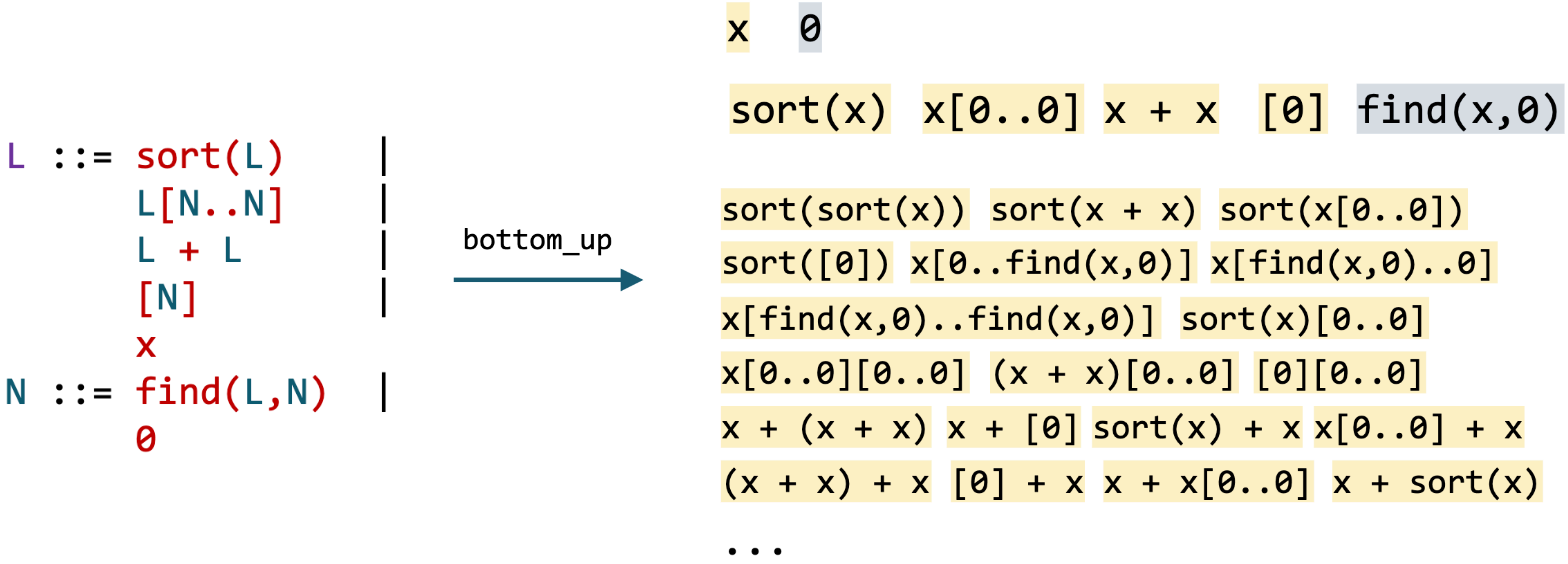
Equivalence reduction

infeasible



Top-down propagation

Equivalent programs



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

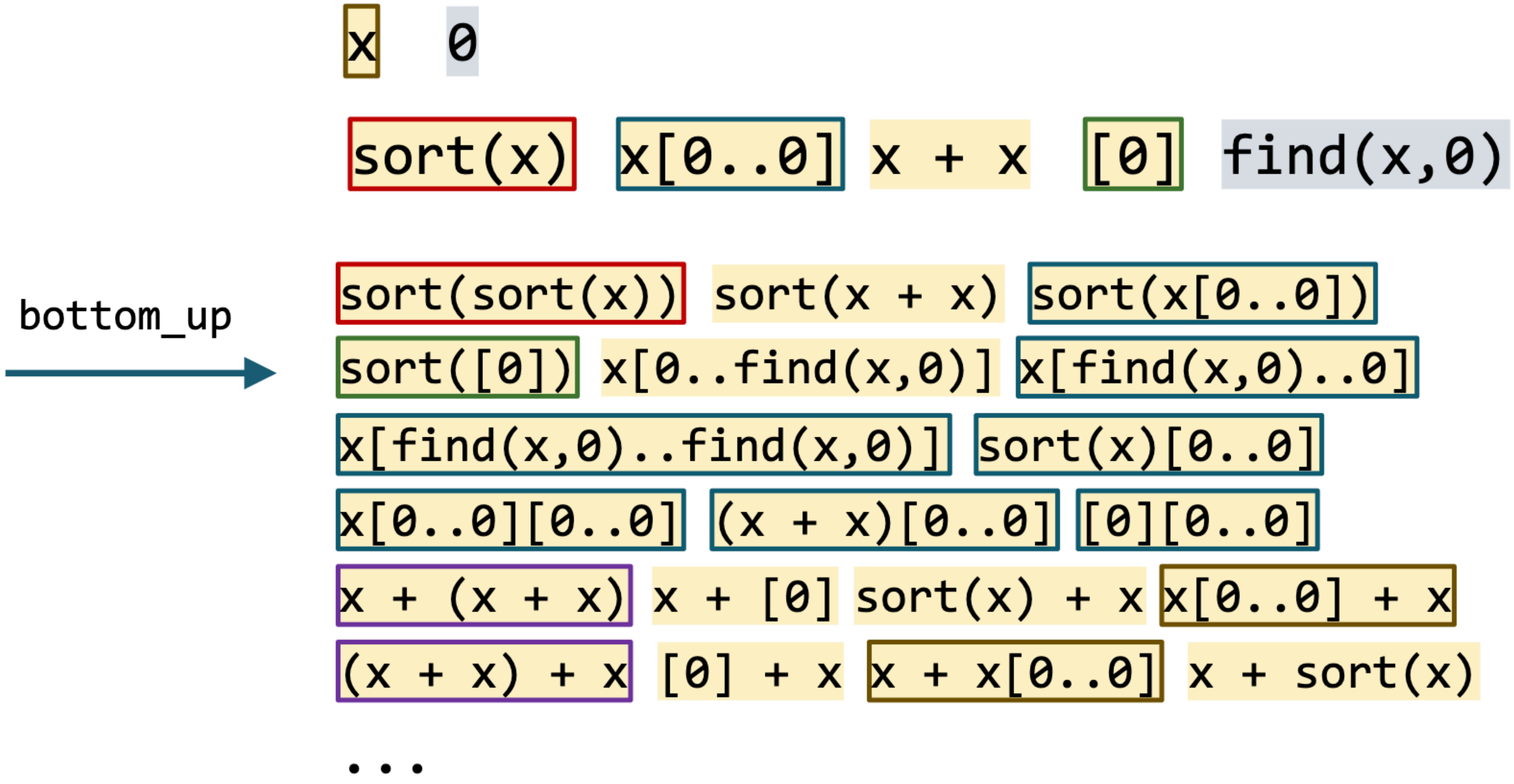
```
x   0

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]

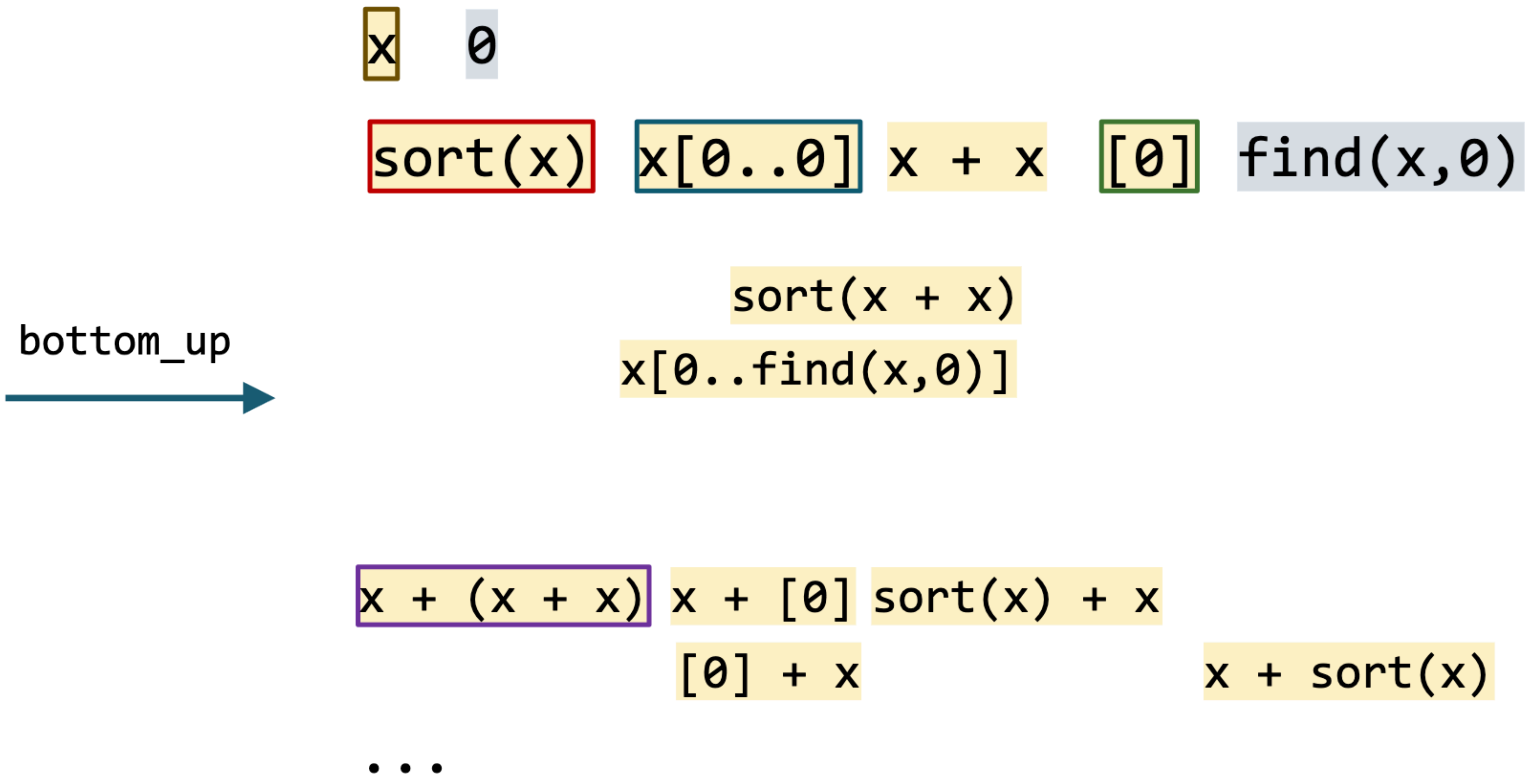
x + (x + x)   x + [0]   sort(x) + x   x[0..0] + x

(x + x) + x   [0] + x   x + x[0..0]   x + sort(x)

...

Equivalent programs

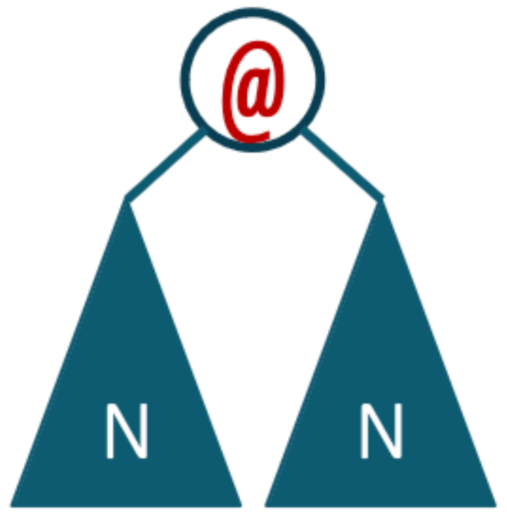
```
L ::= sort(L)     |
      L[N..N]     |
      L + L       |
      [N]         |
      x
N ::= find(L,N)   |
      0
```



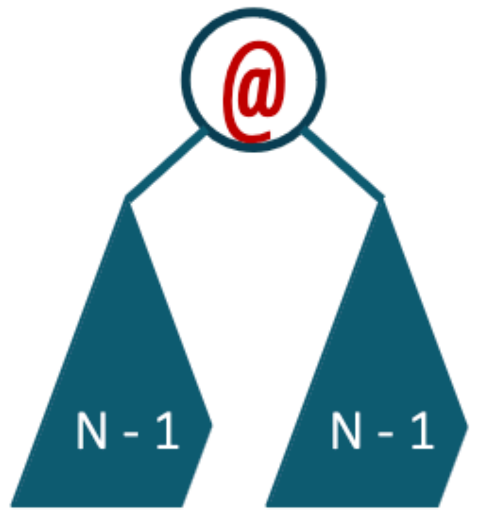
Bottom-up + equivalence reduction

```
bottom-up (<$\Sigma$, 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($t_1$,…,$t_k$)
```



$m * N^2$



$m * (N - 1)^2$

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!

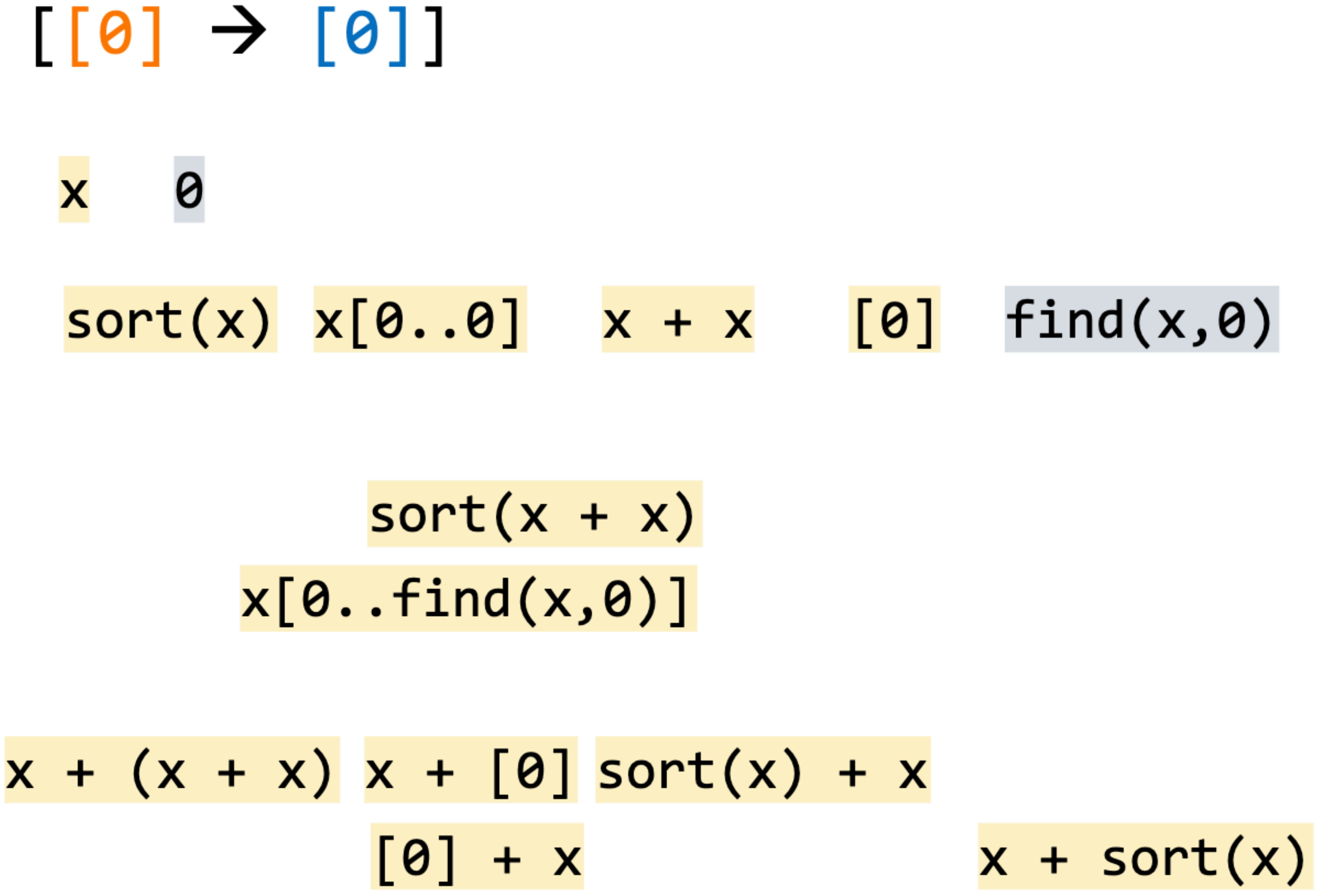
Observational equivalence

```
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`

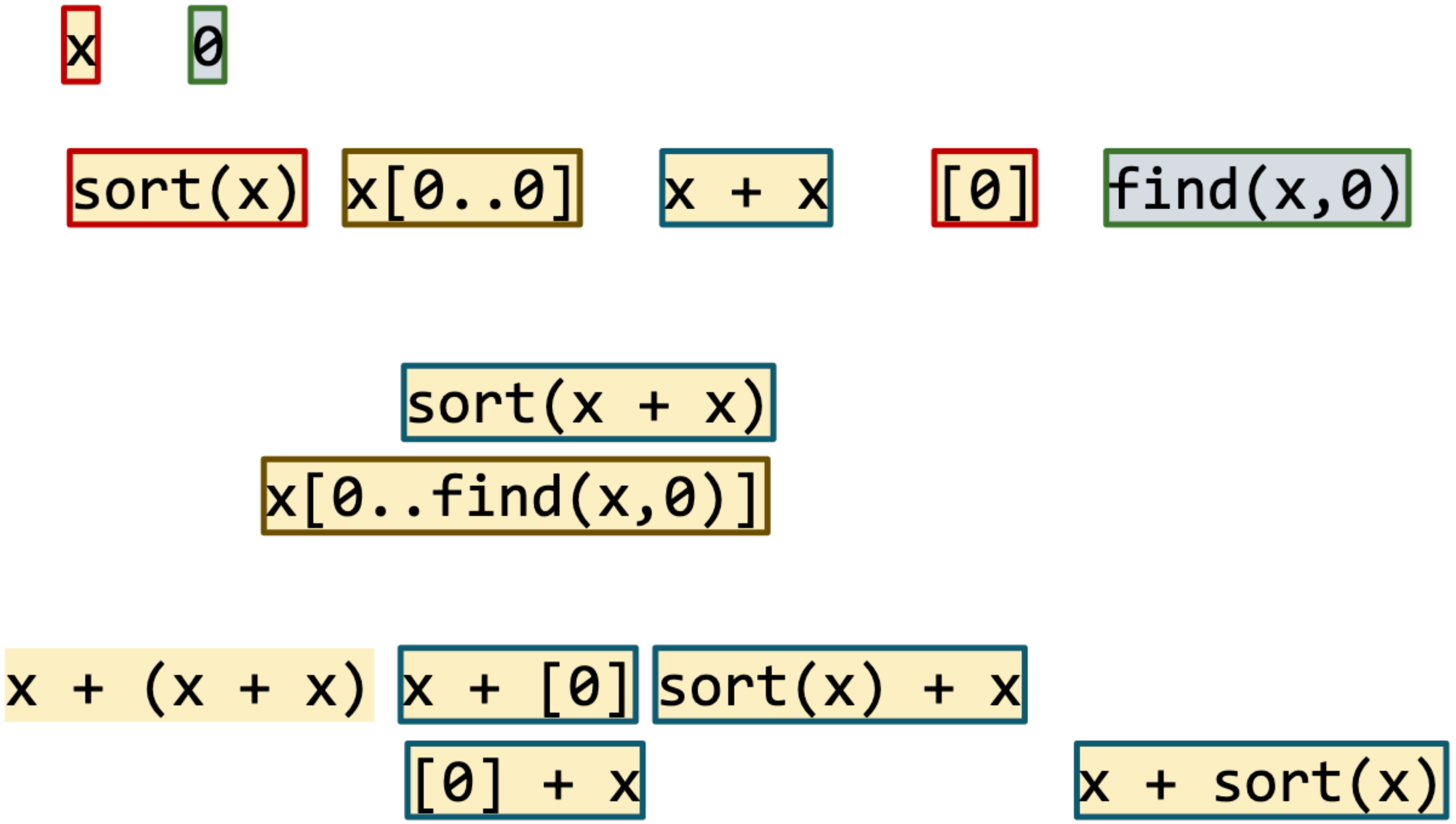
`[0] + x` `x + sort(x)`

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`

how to implement the reduction efficiently?

`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)
- **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?

Other Strategies: Synthesis Through Unification

- Idea: Solve many simpler problems, combine their solutions.
- Rajeev Alur, Pavol Černý, Arjun Radhakrishna, Synthesis Through Unification, 2015
- STUN provides a general framework for breaking down a global search into a series of local searches.