#4: Equivalence Reduction

Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

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

```
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

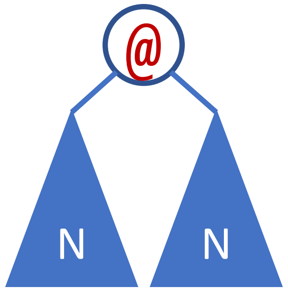
Candidates are complete but might not be whole

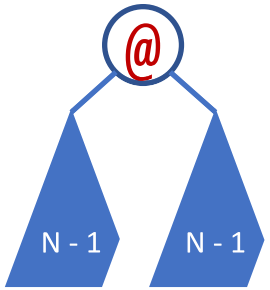
- Can always run on inputs
- Cannot always relate to outputs

How to make it scale

Prune

- Discard useless subprograms





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

Prioritize

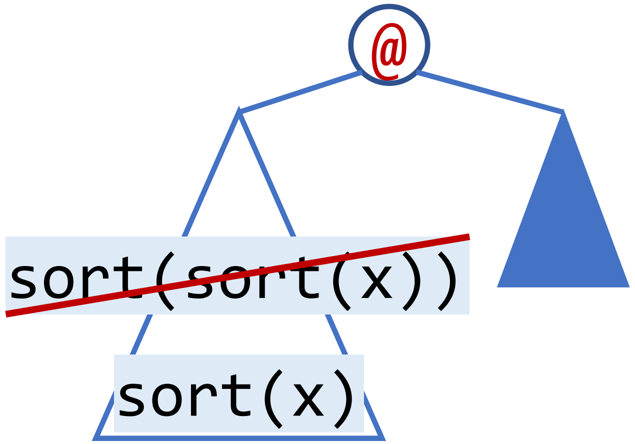
- Explore promising candidates first

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

dequeue this first

When can we discard a subprogram?

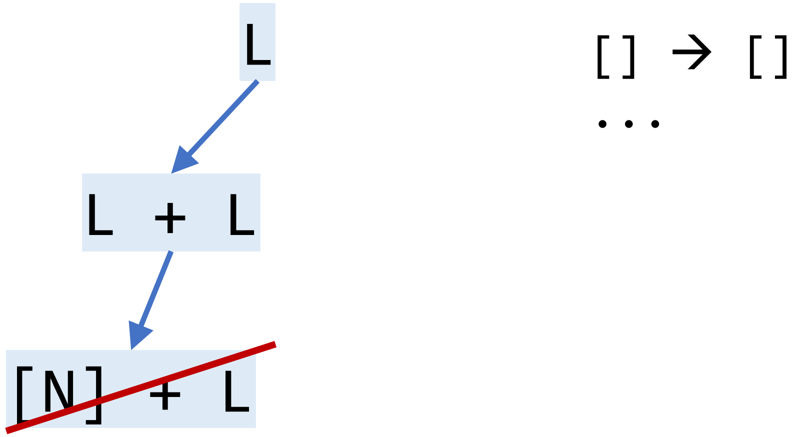
redundant



Equivalence reduction

(also: symmetry breaking)

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

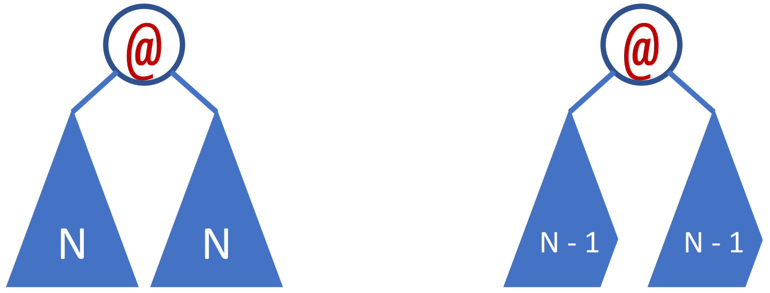
```
L ::= sort(L)     |
      L[N..N]     |
      L + L       |
      [N]         |
      x
N ::= find(L,N)   |
      0
```

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 (<T, N, R, S>, [i → o]) {
  bank := [t | A ::= t in R]
  while (true)
    forall (p in bank)
      if (p([i]) = [o])
        return p;
    bank += grow(bank);
}

grow (bank) {
  bank' := []
  forall (A ::= rhs in R)
    bank' += [rhs[B -> p] | p in bank, B →* p]
  return [p' in bank' | forall p in bank: !equiv(p, p')];
}
```



$m * N^2$

$m * (N - 1)^2$

Bottom-up + equivalence reduction

```
bottom-up (<T, N, R, S>, [i → o]) {
  bank := [t | A ::= t in R]
  while (true)
    forall (p in bank)
      if (p([i]) = [o])
        return p;
    bank += grow(bank);
}

grow (bank) {
  bank' := []
  forall (A ::= rhs in R)
    bank' += [rhs[B -> p] | p in bank, B →* p]
  return [p' in bank' | forall p in bank: !equiv(p, 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)

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-specifies 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:
  - **λ²** [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?