Type-Targeted Testing

Eric Seidel, Niki Vazou, Ranjit Jhala
UC San Diego

```
data Tree
  = Leaf
  | Node Int Tree Tree

insert :: Int -> Tree -> Tree
delete :: Int -> Tree -> Tree
```

```haskell
data Tree
  = Leaf
  | Node Int Tree Tree

insert :: Int -> Tree -> Tree
delete :: Int -> Tree -> Tree
```

Did I get it "right"?

An Oracle for Testing

```
isBST t = case t of
  Leaf        -> True
  Node x l r -> abs (height l - height r) <= 1
             && all (< x) l && all (> x) r
             && isBST l     && isBST r
```

QuickCheck

```
prop_insert_bst x t
  = isBST t ==> isBST (insert x t)
```

```
>>> quickCheck prop_insert_bst
+++ OK, passed 100 tests.
```

QuickCheck

```
prop_insert_bst x t
  = isBST t ==> isBST (insert x t)
```

```
>>> quickCheck prop_insert_bst
+++ OK, passed 100 tests.
```

How is this possible?
Valid trees are a **sparse subset** of all trees!

QuickCheck With Statistics

```
prop_insert_bst x t
  = isBST t
    ==> collect (size t)
                (isBST (insert x t))
```

```
>>> quickCheck prop_insert_bst
+++ OK, passed 100 tests:
73% 0
21% 1
 6% 2
```

QuickCheck With Statistics

```
prop_insert_bst x t
  = isBST t
    ==> collect (size t)
                (isBST (insert x t))
```

```
>>> quickCheck prop_insert_bst
+++ OK, passed 100 tests:
73% 0
21% 1
 6% 2
```

73% of trees were **empty** and
21% only had one element

QuickCheck: Non-Trivial Trees

```
prop_insert_bst x t
  = isBST t && size t > 1
    ==> isBST (insert x t)
```

```
>>> quickCheck prop_insert_bst
*** Gave up! Passed only 37 tests.
```

QuickCheck: Non-Trivial Trees

```
prop_insert_bst x t
  = isBST t && size t > 1
    ==> isBST (insert x t)
```

```
>>> quickCheck prop_insert_bst
*** Gave up! Passed only 37 tests.
```

Less than 10% of generated trees were valid

Custom Generators

```
newtype BST = BST Tree

instance Arbitrary BST where
  arbitrary = ...

prop_insert_bst x (BST t) = ...
```

Custom Generators

```
newtype BST = BST Tree

instance Arbitrary BST where
  arbitrary = ...

prop_insert_bst x (BST t) = ...
```

Must repeat for any further restrictions on input domain!

SmallCheck

```
prop_insert_bst x t
  = isBST t ==> isBST (insert x t)
```

```
>>> smallCheck 3 prop_insert_bst
Completed 567 tests without failure.
But 434 did not meet ==> condition.
```

SmallCheck

```
prop_insert_bst x t
  = isBST t ==> isBST (insert x t)
```

```
>>> smallCheck 3 prop_insert_bst
Completed 567 tests without failure.
But 434 did not meet ==> condition.
```

Only **133** input trees were valid!

SmallCheck: How Small?

```
prop_insert_bst x t
  = isBST t ==> isBST (insert x t)
```

```
>>> smallCheck 4 prop_insert_bst
....................................
.............................
```

SmallCheck: How Small?

```
prop_insert_bst x t
  = isBST t ==> isBST (insert x t)
```

```
>>> smallCheck 4 prop_insert_bst
....................................
..............................
```

Exponential blowup in input space
confines search to **very small** inputs

How can we **systematically** generate only **valid** inputs?

Target

Generates tests from **refinement types** via query-decode-check loop

1. Translate input types into SMT **query**

2. **Decode** SMT model into concrete values

3. Run function and **check** that result inhabits output type

Refinement Types

```
{v:t | p}
```

The set of values `v` of type `t` satisfying a predicate `p`

Refinement Types

```
type Nat   = {v:Int | 0 <= v}
type Pos   = {v:Int | 0 <  v}
type Rng N = {v:Int | 0 <= v && v < N}
```

The natural numbers, positive integers,
and integers in a range

Refinement Types

```
x:Nat -> {v:Nat | v = x + 1}
```

Functions that take a natural number
and increment it by one

Target

Generates tests from **refinement types** via query-decode-check loop

1. Translate input types into SMT **query**
2. **Decode** SMT model into concrete values
3. Run function and **check** that result inhabits output type

Step 1: Query

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Step 1: Query

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

Step 1: Query

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

$$\mathsf{C}_0 \doteq \; 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1$$

Step 2: Decode

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

$\mathsf{C}_0 \doteq 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1$

A model $[\mathrm{r}_1 \mapsto 1, \mathrm{r}_2 \mapsto 1, \mathrm{s} \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 1 0
```

Step 3: Check

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

$$C_0 \doteq \ 0 \leq r_1 \wedge 0 \leq r_2 \wedge 0 \leq s < r_1$$

A model $[r_1 \mapsto 1, r_2 \mapsto 1, s \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 1 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

Step 3: Check

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

$$C_0 \doteq \ 0 \leq r_1 \wedge 0 \leq r_2 \wedge 0 \leq s < r_1$$

A model $[r_1 \mapsto 1, r_2 \mapsto 1, s \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 1 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

After substituting `v` and `r2`: $0 \leq 0 \quad \wedge \quad 0 < 1$

Step 3: Check

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

$$C_0 \doteq \ 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1$$

A model $[\mathrm{r}_1 \mapsto 1, \mathrm{r}_2 \mapsto 1, \mathrm{s} \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 1 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

After substituting `v` and `r2`: $0 \leq 0 \quad \wedge \quad 0 < 1$ **VALID**

Step 3: Check

```
type Nat    = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic

$C_0 \doteq \; 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1$

A model $[\mathrm{r}_1 \mapsto 1, \mathrm{r}_2 \mapsto 1, \mathrm{s} \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 1 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

After substituting **`v`** and **`r2`**: $0 \leq 0 \quad \wedge \quad 0 < 1$ **VALID**

Force new test by adding refutation constraint $\neg(\mathrm{r}_1 = 1 \wedge \mathrm{r}_2 = 1 \wedge \mathrm{s} = 0)$

Repeat With New Test

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic, excluding 1st test

$$\mathsf{C}_1 \doteq \; 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1 \wedge \neg(\mathrm{r}_1 = 1 \wedge \mathrm{r}_2 = 1 \wedge \mathrm{s} = 0)$$

Repeat With New Test

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic, excluding 1st test

$$C_1 \doteq 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1 \wedge \neg(\mathrm{r}_1 = 1 \wedge \mathrm{r}_2 = 1 \wedge \mathrm{s} = 0)$$

A model $[\mathrm{r}_1 \mapsto 1, \mathrm{r}_2 \mapsto 0, \mathrm{s} \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 0 0
0
```

Repeat With New Test

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic, excluding 1st test

$$C_1 \doteq \; 0 \leq \mathrm{r}_1 \wedge 0 \leq \mathrm{r}_2 \wedge 0 \leq s < \mathrm{r}_1 \wedge \neg(\mathrm{r}_1 = 1 \wedge \mathrm{r}_2 = 1 \wedge \mathrm{s} = 0)$$

A model $[\mathrm{r}_1 \mapsto 1, \mathrm{r}_2 \mapsto 0, \mathrm{s} \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 0 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

Repeat With New Test

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic, excluding 1st test

$$C_1 \doteq\ 0 \leq r_1 \wedge 0 \leq r_2 \wedge 0 \leq s < r_1 \wedge \neg(r_1 = 1 \wedge r_2 = 1 \wedge s = 0)$$

A model $[r_1 \mapsto 1, r_2 \mapsto 0, s \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 0 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

After substituting **v** and **r2**: $0 \leq 0 \quad \wedge \quad 0 < 0$

Repeat With New Test

```
type Nat    = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic, excluding 1st test

$$C_1 \doteq \; 0 \leq r_1 \wedge 0 \leq r_2 \wedge 0 \leq s < r_1 \wedge \neg(r_1 = 1 \wedge r_2 = 1 \wedge s = 0)$$

A model $[r_1 \mapsto 1, r_2 \mapsto 0, s \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 0 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

After substituting `v` and `r2`: $0 \leq 0 \quad \wedge \quad 0 < 0$ **INVALID**

Repeat With New Test

```
type Nat   = {v:Int | 0 <= v}
type Rng N = {v:Int | 0 <= v && v < N}

rescale :: r1:Nat -> r2:Nat -> s:Rng r1 -> Rng r2
```

Represent preconditions directly in logic, excluding 1st test

$$C_1 \doteq \; 0 \leq r_1 \wedge 0 \leq r_2 \wedge 0 \leq s < r_1 \wedge \neg(r_1 = 1 \wedge r_2 = 1 \wedge s = 0)$$

A model $[r_1 \mapsto 1, r_2 \mapsto 0, s \mapsto 0]$ maps to a concrete test case

```
>>> rescale 1 0 0
0
```

Postcondition is: `{v:Int | 0 <= v && v < r2}`

After substituting `v` and `r2`: $0 \leq 0 \quad \wedge \quad 0 < 0$ **INVALID**

`rescale 1 0 0` fails the postcondition check!

Target

Target

Generates tests from refinement types via query-decode-check loop

1. Translate input types into SMT **query**
2. **Decode** SMT model into concrete values
3. Run function and **check** that result inhabits output type

Target

Generates tests from refinement types via query-decode-check loop

1. Translate input types into SMT **query**
2. **Decode** SMT model into concrete values
3. Run function and **check** that result inhabits output type

How should we handle **structured data**?

Containers

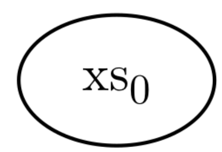
```
type Weight = Pos
type Score  = Rng 100

average :: [(Weight, Score)] -> Score
```

How to generate lists via SMT solver?

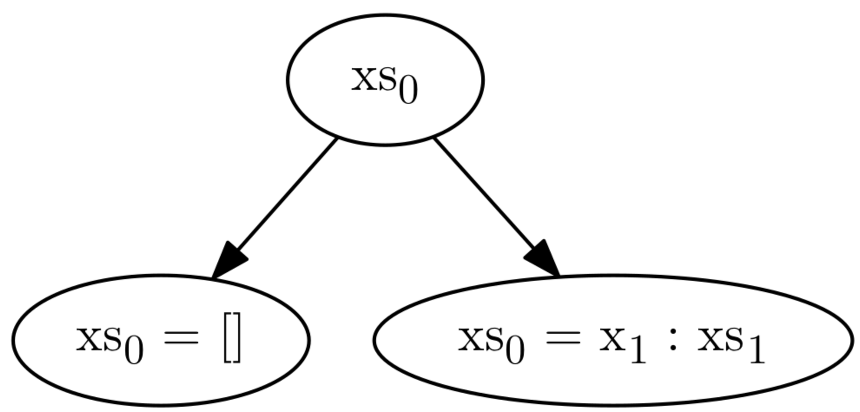
Containers: Query

A **single** set of constraints describes **all possible** inputs



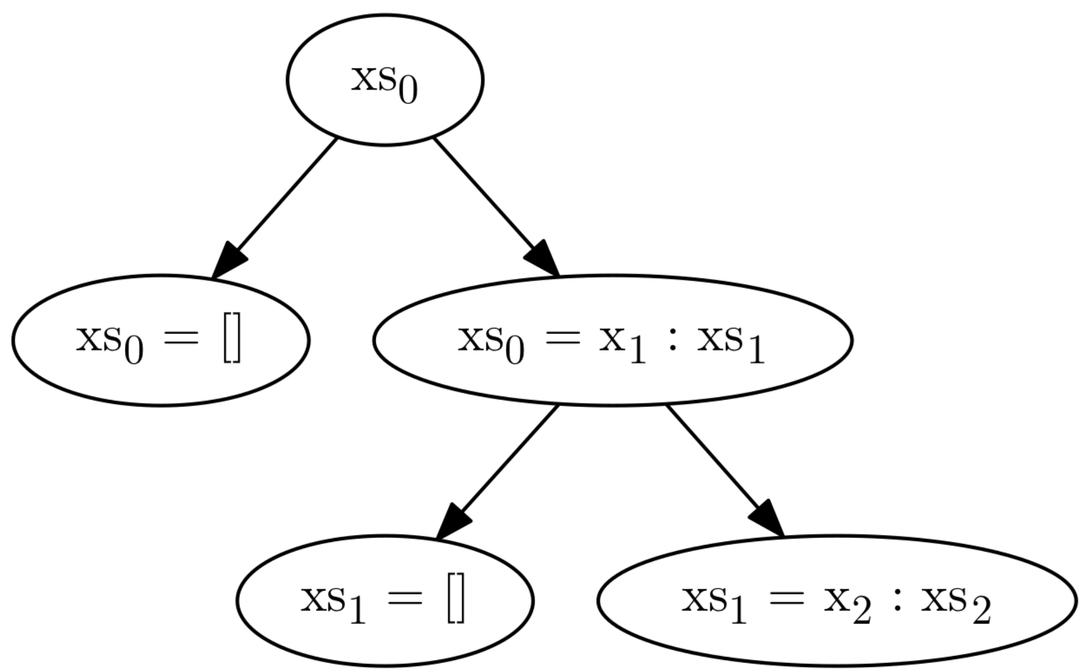
Containers: Query

A **single** set of constraints describes **all possible** inputs



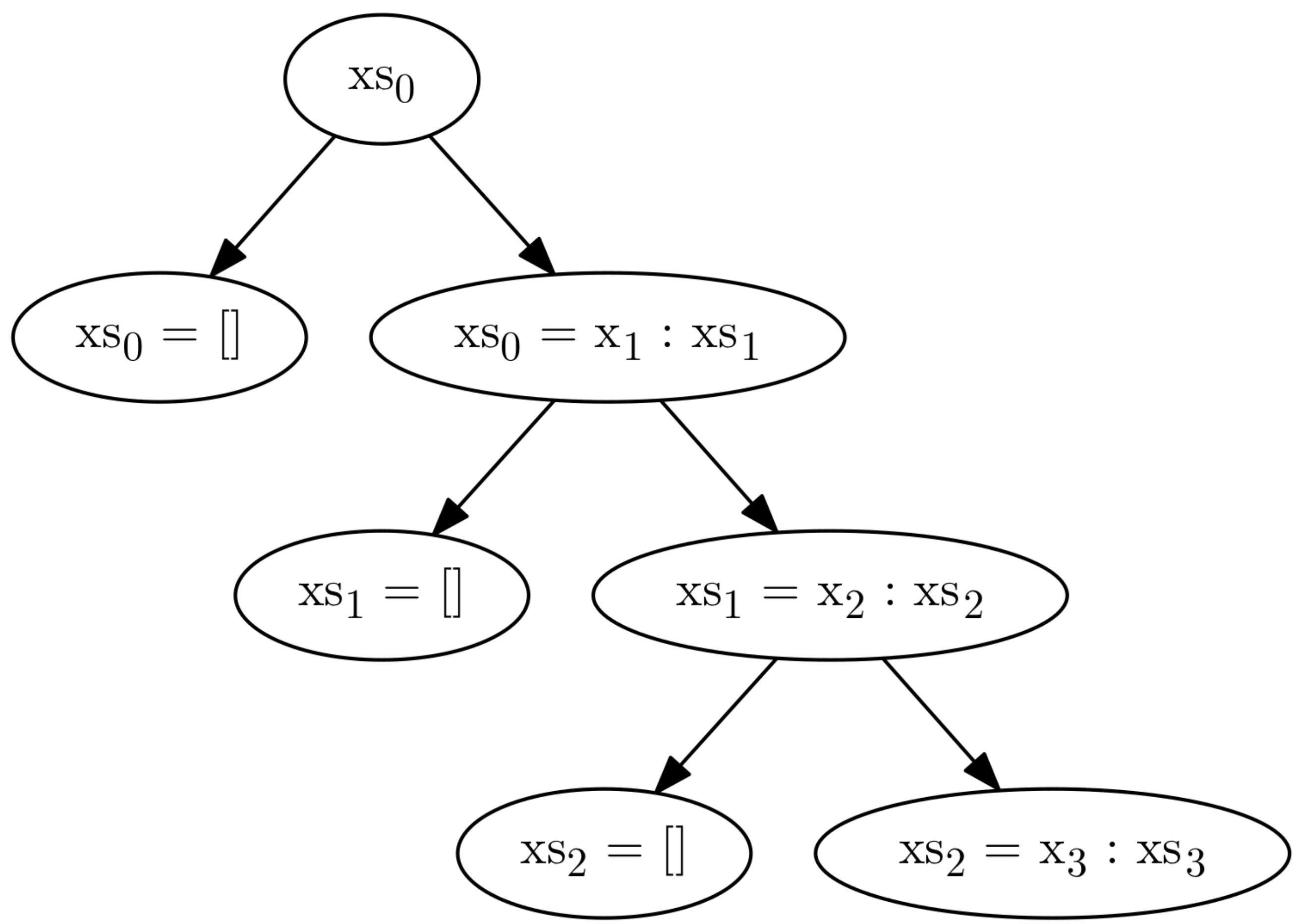
Containers: Query

A **single** set of constraints describes **all possible** inputs



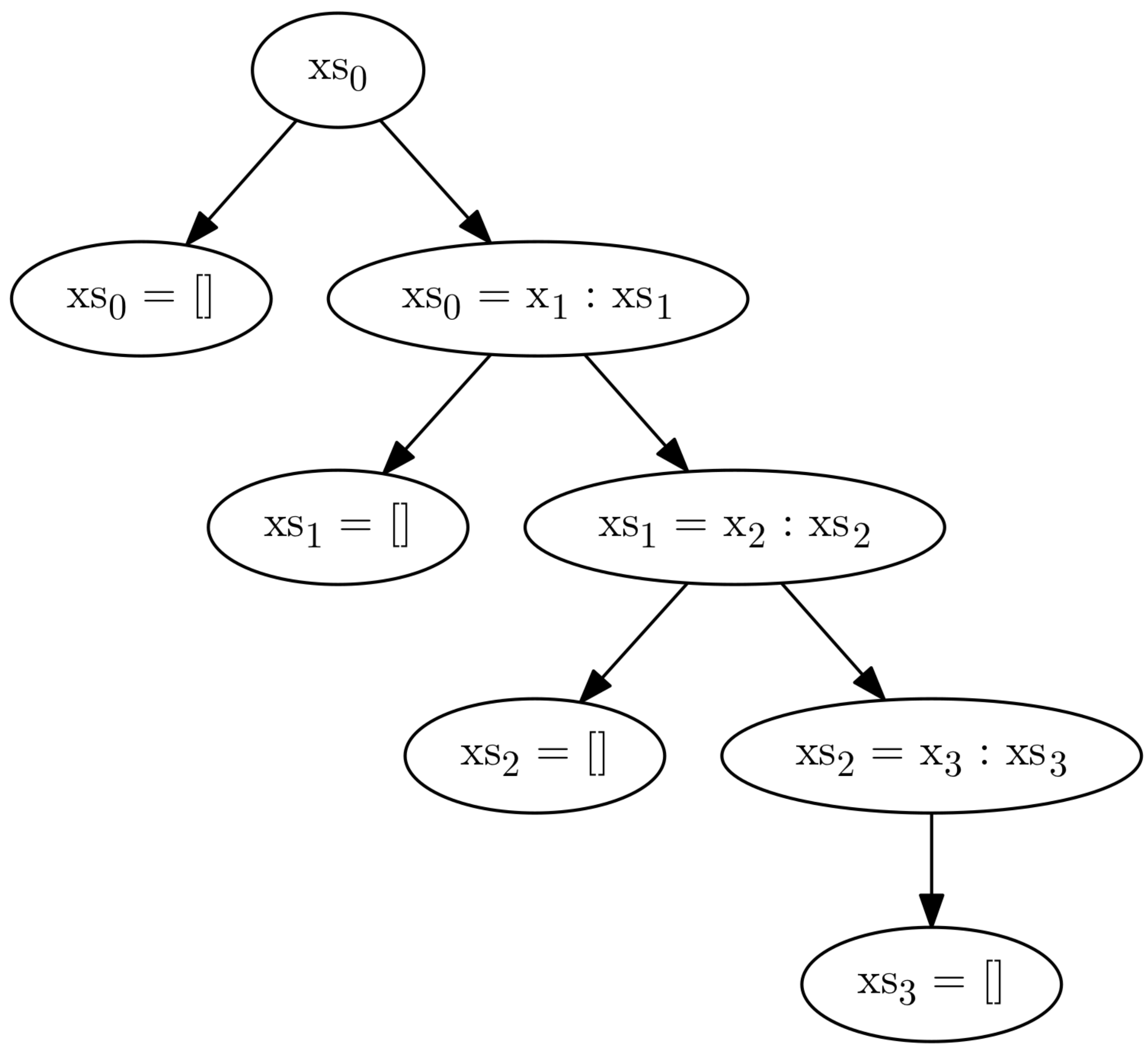
Containers: Query

A **single** set of constraints describes **all possible** inputs



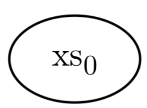
Containers: Query

A **single** set of constraints describes **all possible** inputs



Containers: Query

A **single** set of constraints describes **all possible** inputs

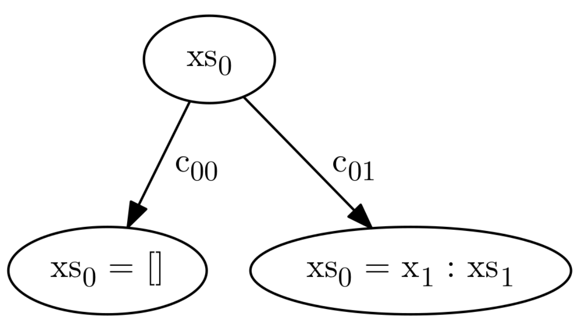


Choice variables *c* **guard** other constraints

$C_{\mathsf{list}} \doteq$

Containers: Query

A **single** set of constraints describes **all possible** inputs

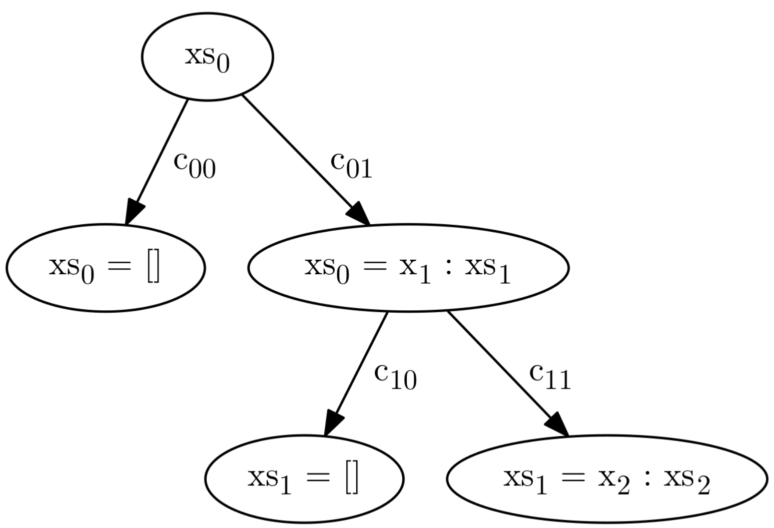


Choice variables *c* **guard** other constraints

$$C_{\text{list}} \doteq (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \wedge (c_{00} \oplus c_{01})$$

Containers: Query

A **single** set of constraints describes **all possible** inputs

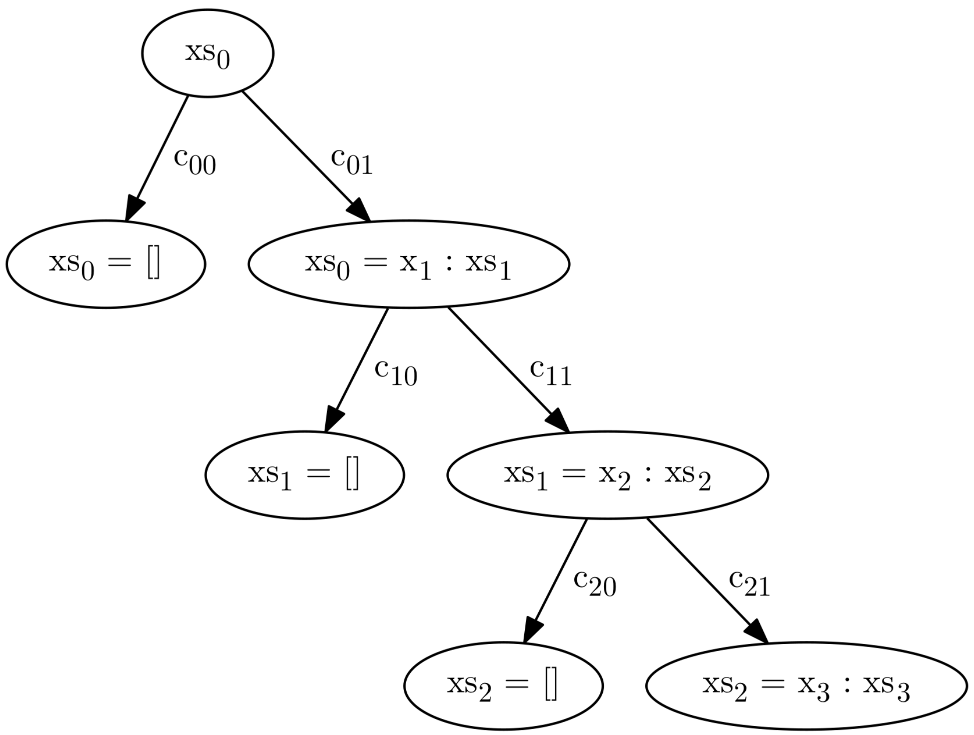


Choice variables *c* **guard** other constraints

$$
\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (\mathrm{c}_{00} \Rightarrow \mathrm{xs}_0 = []) \wedge (\mathrm{c}_{01} \Rightarrow \mathrm{xs}_0 = \mathrm{x}_1 : \mathrm{xs}_1) \wedge (\mathrm{c}_{00} \oplus \mathrm{c}_{01}) \\
& \wedge (\mathrm{c}_{10} \Rightarrow \mathrm{xs}_1 = []) \wedge (\mathrm{c}_{11} \Rightarrow \mathrm{xs}_1 = \mathrm{x}_2 : \mathrm{xs}_2) \wedge (\mathrm{c}_{01} \Rightarrow \mathrm{c}_{10} \oplus \mathrm{c}_{11})
\end{aligned}
$$

Containers: Query

A **single** set of constraints describes **all possible** inputs

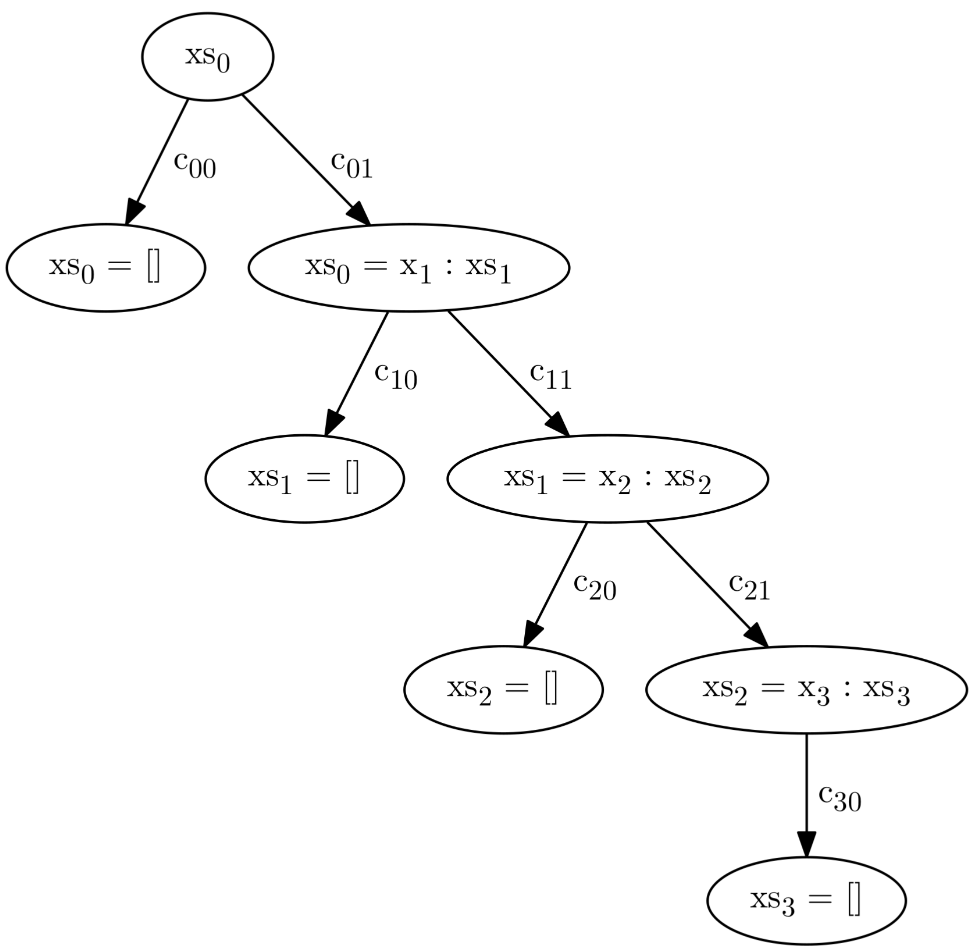


Choice variables *c* **guard** other constraints

$$
\begin{aligned}
C_{\text{list}} \doteq\ & (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \wedge (c_{00} \oplus c_{01}) \\
& \wedge (c_{10} \Rightarrow xs_1 = []) \wedge (c_{11} \Rightarrow xs_1 = x_2 : xs_2) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \\
& \wedge (c_{20} \Rightarrow xs_2 = []) \wedge (c_{21} \Rightarrow xs_2 = x_3 : xs_3) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21})
\end{aligned}
$$

Containers: Query

A **single** set of constraints describes **all possible** inputs

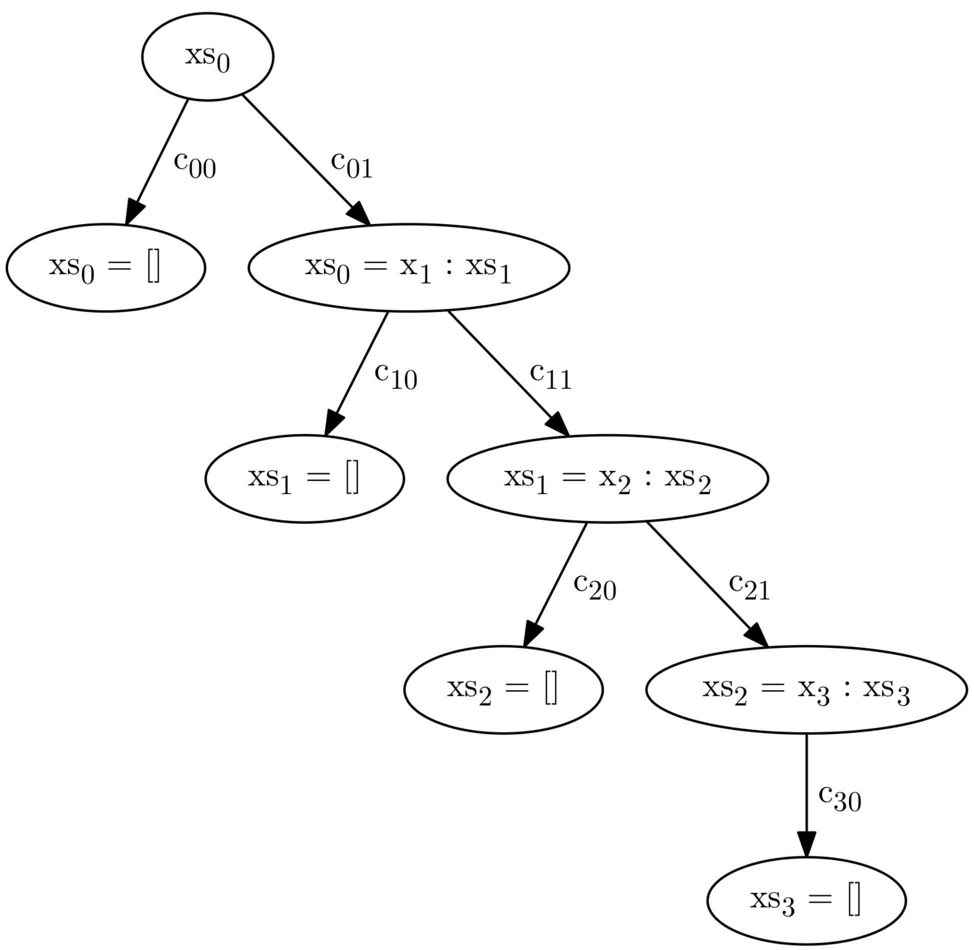


Choice variables *c* **guard** other constraints

$$
\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (\mathrm{c}_{00} \Rightarrow \mathrm{xs}_0 = []) \land (\mathrm{c}_{01} \Rightarrow \mathrm{xs}_0 = \mathrm{x}_1 : \mathrm{xs}_1) \land (\mathrm{c}_{00} \oplus \mathrm{c}_{01}) \\
& \land (\mathrm{c}_{10} \Rightarrow \mathrm{xs}_1 = []) \land (\mathrm{c}_{11} \Rightarrow \mathrm{xs}_1 = \mathrm{x}_2 : \mathrm{xs}_2) \land (\mathrm{c}_{01} \Rightarrow \mathrm{c}_{10} \oplus \mathrm{c}_{11}) \\
& \land (\mathrm{c}_{20} \Rightarrow \mathrm{xs}_2 = []) \land (\mathrm{c}_{21} \Rightarrow \mathrm{xs}_2 = \mathrm{x}_3 : \mathrm{xs}_3) \land (\mathrm{c}_{11} \Rightarrow \mathrm{c}_{20} \oplus \mathrm{c}_{21}) \\
& \land (\mathrm{c}_{30} \Rightarrow \mathrm{xs}_3 = []) \land (\mathrm{c}_{21} \Rightarrow \mathrm{c}_{30})
\end{aligned}
$$

Containers: Query

A **single** set of constraints describes **all possible** inputs



Choice variables *c* **guard** other constraints

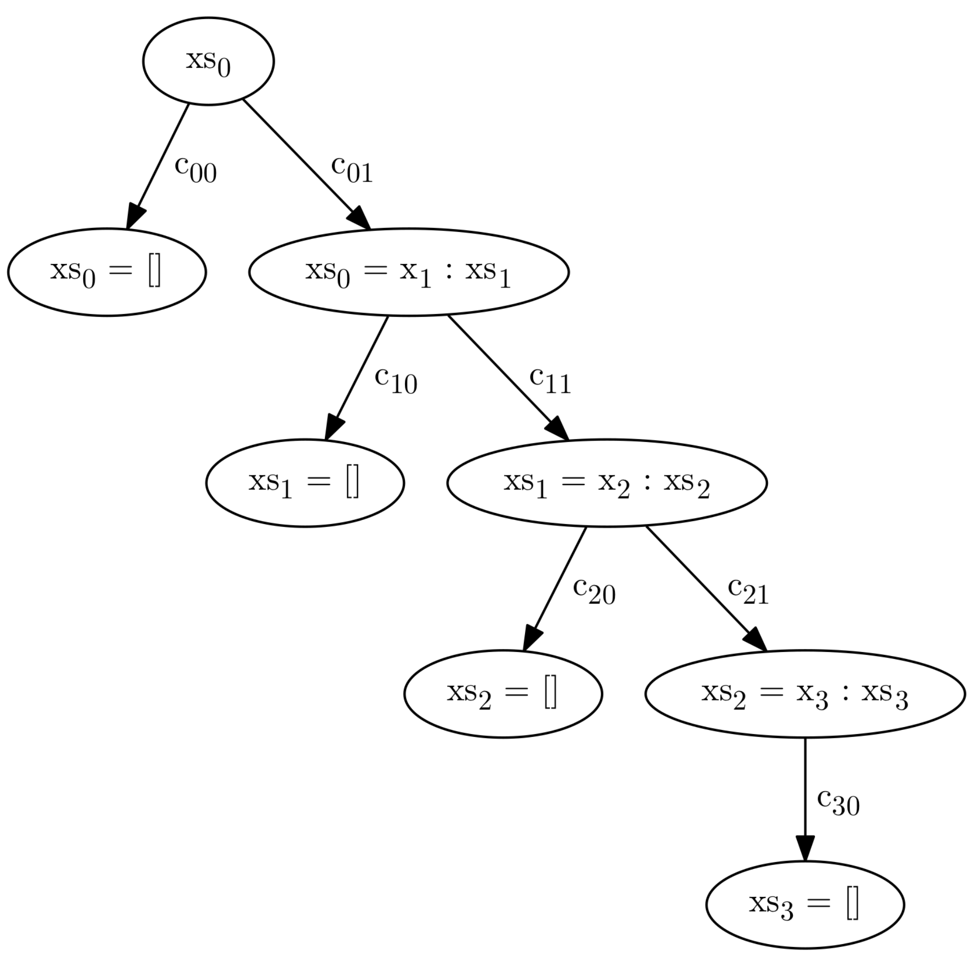
$$
\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (c_{00} \Rightarrow \mathrm{xs}_0 = []) \wedge (c_{01} \Rightarrow \mathrm{xs}_0 = \mathrm{x}_1 : \mathrm{xs}_1) \wedge (c_{00} \oplus c_{01}) \\
& \wedge (c_{10} \Rightarrow \mathrm{xs}_1 = []) \wedge (c_{11} \Rightarrow \mathrm{xs}_1 = \mathrm{x}_2 : \mathrm{xs}_2) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \\
& \wedge (c_{20} \Rightarrow \mathrm{xs}_2 = []) \wedge (c_{21} \Rightarrow \mathrm{xs}_2 = \mathrm{x}_3 : \mathrm{xs}_3) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21}) \\
& \wedge (c_{30} \Rightarrow \mathrm{xs}_3 = []) \wedge (c_{21} \Rightarrow c_{30})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C}_{\mathsf{data}} \doteq\ & (c_{01} \Rightarrow \mathrm{x}_1 = (\mathrm{w}_1, \mathrm{s}_1) \ \wedge\ 0 < \mathrm{w}_1 \ \wedge\ 0 \le \mathrm{s}_1 < 100) \\
& \wedge (c_{11} \Rightarrow \mathrm{x}_2 = (\mathrm{w}_2, \mathrm{s}_2) \ \wedge\ 0 < \mathrm{w}_2 \ \wedge\ 0 \le \mathrm{s}_2 < 100) \\
& \wedge (c_{21} \Rightarrow \mathrm{x}_3 = (\mathrm{w}_3, \mathrm{s}_3) \ \wedge\ 0 < \mathrm{w}_3 \ \wedge\ 0 \le \mathrm{s}_3 < 100)
\end{aligned}
$$

Full constraint $\quad \mathsf{C} \doteq\ \mathsf{C}_{\mathsf{list}} \wedge \mathsf{C}_{\mathsf{data}}$

Containers: Query

A **single** set of constraints describes **all possible** inputs

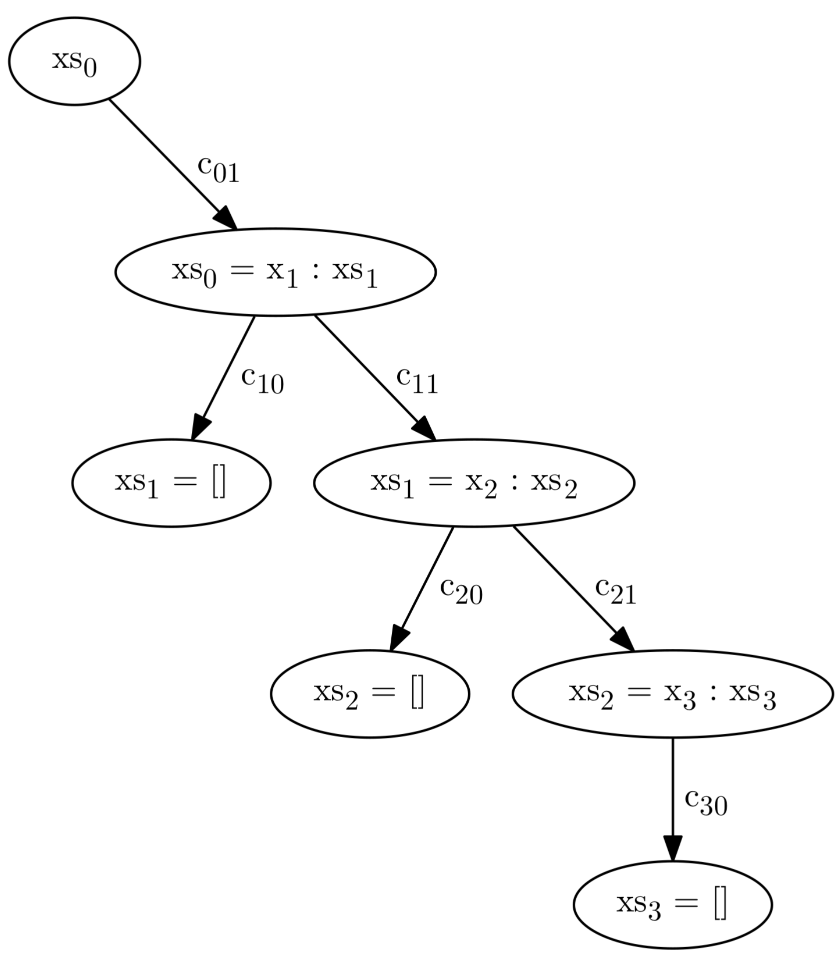


Follow choice variables to reconstruct the list

$$[\ \mathrm{c}_{00} \mapsto \ \mathsf{false},\ \mathrm{c}_{01} \mapsto \ \mathsf{true},\ \mathrm{x}_1 \mapsto (\mathrm{w}_1, \mathrm{s}_1),\ \mathrm{w}_1 \mapsto 1,\ \mathrm{s}_1 \mapsto 2,$$
$$\mathrm{c}_{10} \mapsto \ \mathsf{true},\ \mathrm{c}_{11} \mapsto \ \mathsf{false},\ \mathrm{x}_2 \mapsto (\mathrm{w}_2, \mathrm{s}_2),\ \mathrm{w}_2 \mapsto 3,\ \mathrm{s}_2 \mapsto 4, \ldots\ ]$$

Containers: Query

A **single** set of constraints describes **all possible** inputs



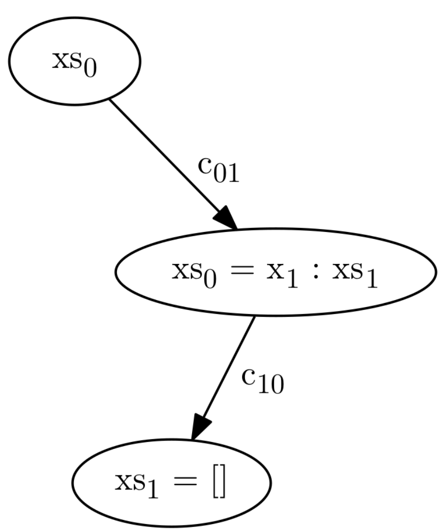
Follow choice variables to reconstruct the list

$$[\ c_{00} \mapsto \textsf{false},\ c_{01} \mapsto \textsf{true},\ x_1 \mapsto (w_1, s_1),\ w_1 \mapsto 1,\ s_1 \mapsto 2,$$
$$c_{10} \mapsto \textsf{true},\ c_{11} \mapsto \textsf{false},\ x_2 \mapsto (w_2, s_2),\ w_2 \mapsto 3,\ s_2 \mapsto 4, \ldots\ ]$$

$$c_{01} \mapsto \textsf{true} \Rightarrow \mathrm{xs}_0 = \mathrm{x}_1 : \mathrm{xs}_1$$

Containers: Query

A **single** set of constraints describes **all possible** inputs



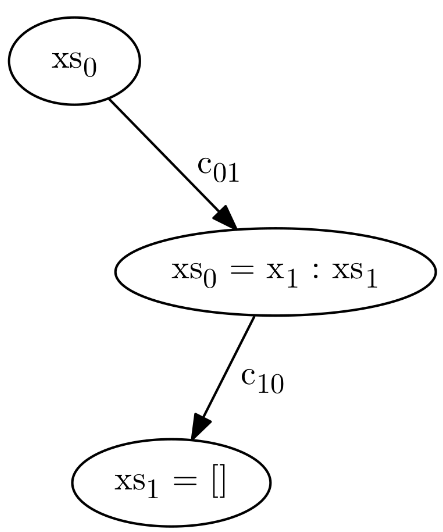
Follow choice variables to reconstruct the list

$$[\ \mathrm{c}_{00} \mapsto \mathsf{false},\ \mathrm{c}_{01} \mapsto \mathsf{true},\ \mathrm{x}_1 \mapsto (\mathrm{w}_1, \mathrm{s}_1),\ \mathrm{w}_1 \mapsto 1,\ \mathrm{s}_1 \mapsto 2,$$
$$\mathrm{c}_{10} \mapsto \mathsf{true},\ \mathrm{c}_{11} \mapsto \mathsf{false},\ \mathrm{x}_2 \mapsto (\mathrm{w}_2, \mathrm{s}_2),\ \mathrm{w}_2 \mapsto 3,\ \mathrm{s}_2 \mapsto 4, \ldots\ ]$$

$$\mathrm{c}_{01} \mapsto \mathsf{true} \Rightarrow \mathrm{xs}_0 = \mathrm{x}_1 : \mathrm{xs}_1$$
$$\mathrm{c}_{10} \mapsto \mathsf{true} \Rightarrow \mathrm{xs}_1 = []$$

Containers: Query

A **single** set of constraints describes **all possible** inputs



Follow choice variables to reconstruct the list

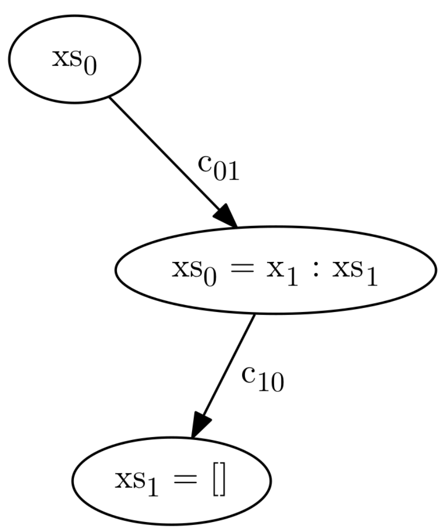
$$[\ c_{00} \mapsto \ \mathsf{false},\ c_{01} \mapsto \ \mathsf{true},\ x_1 \mapsto (w_1, s_1),\ w_1 \mapsto 1,\ s_1 \mapsto 2,$$
$$c_{10} \mapsto \ \mathsf{true},\ c_{11} \mapsto \ \mathsf{false},\ x_2 \mapsto (w_2, s_2),\ w_2 \mapsto 3,\ s_2 \mapsto 4, \ldots\ ]$$

$$c_{01} \mapsto \ \mathsf{true} \Rightarrow \mathrm{xs}_0 = x_1 : \mathrm{xs}_1$$
$$c_{10} \mapsto \ \mathsf{true} \Rightarrow \mathrm{xs}_1 = []$$

Realized value: `[(1,2)]`

Containers: Query

A **single** set of constraints describes **all possible** inputs



Follow choice variables to reconstruct the list

$$[\ c_{00} \mapsto \textsf{false},\ c_{01} \mapsto \textsf{true},\ x_1 \mapsto (w_1, s_1),\ w_1 \mapsto 1,\ s_1 \mapsto 2,$$
$$c_{10} \mapsto \textsf{true},\ c_{11} \mapsto \textsf{false},\ x_2 \mapsto (w_2, s_2),\ w_2 \mapsto 3,\ s_2 \mapsto 4, \ldots\ ]$$

$$c_{01} \mapsto \textsf{true} \Rightarrow xs_0 = x_1 : xs_1$$
$$c_{10} \mapsto \textsf{true} \Rightarrow xs_1 = []$$

Realized value: `[(1,2)]`

Only refute constraints that contribute to **realized** value

$$\neg(c_{00} = \textsf{false} \wedge c_{01} = \textsf{true} \wedge x_1 = (w_1, s_1) \wedge w_1 = 1 \wedge s_1 = 2 \wedge c_{10} = \textsf{true})$$

Structured Containers

```
best :: k:Nat -> {xs:[Score] | k <= len xs}
     -> {v:[Score] | k = len v}
```

`best` takes a list of **at least** `k` scores,
and returns a list with **exactly** `k` scores

Structured Containers

```
best :: k:Nat -> {xs:[Score] | k <= len xs}
     -> {v:[Score] | k = len v}
```

`best` takes a list of **at least** `k` scores,
and returns a list with **exactly** `k` scores

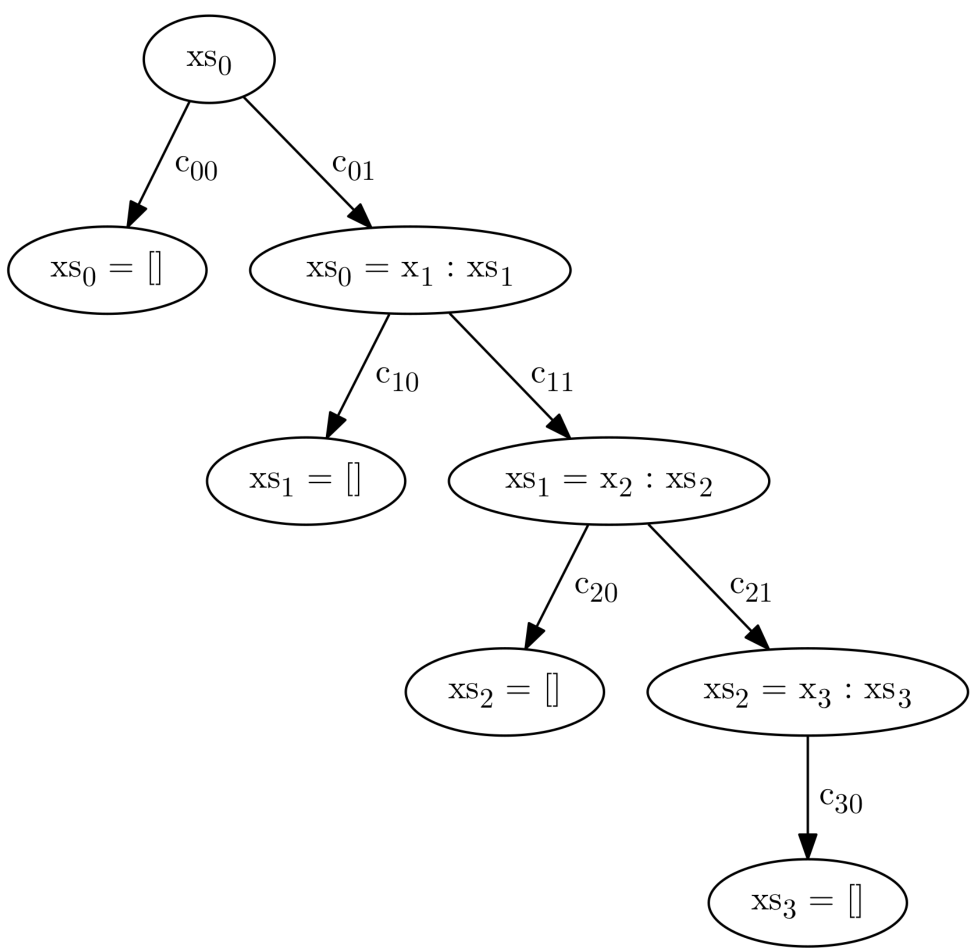
```
measure len :: [a] -> Nat
len []      = 0
len (x:xs)  = 1 + len xs
```

`len` is a **logical function** that describes the length of a list.

We instantiate measure definition each time we unfold `[]` or `(:)`

Structured Containers

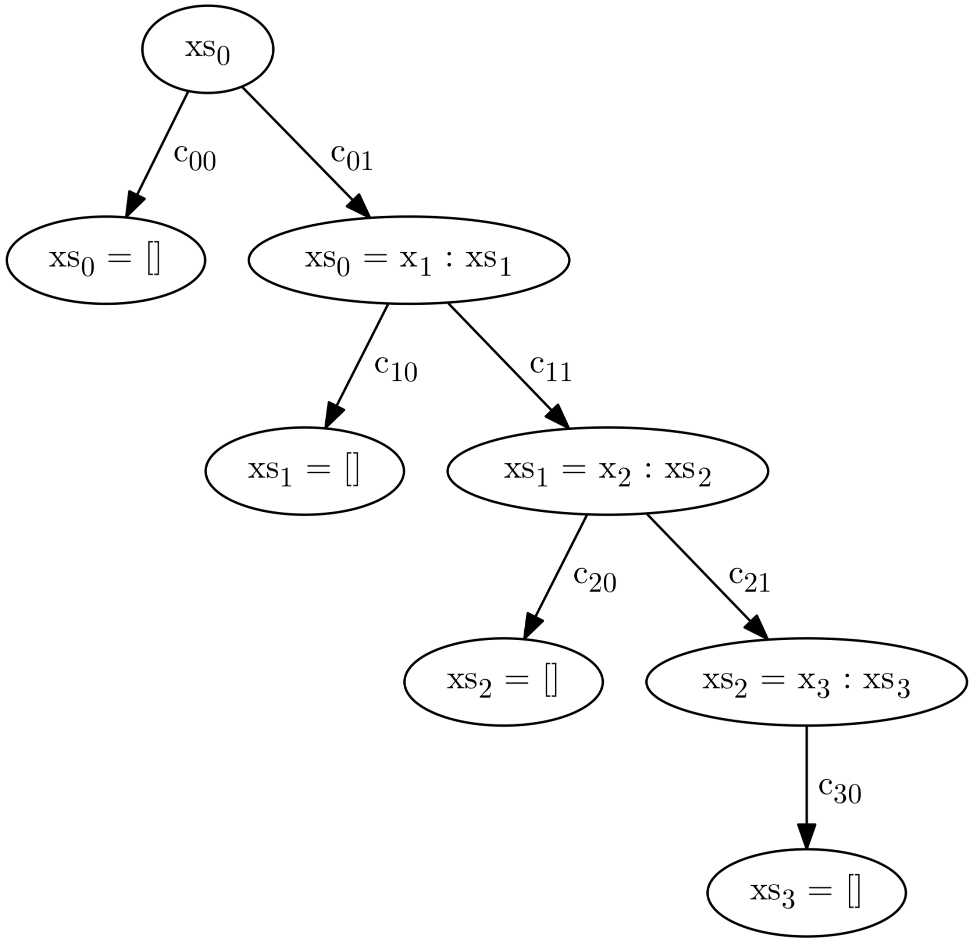
```
best :: k:Nat -> {xs:[Score] | k <= len xs}
     -> {v:[Score] | k = len v}
```



$$
\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \wedge (c_{00} \oplus c_{01}) \\
& \wedge (c_{10} \Rightarrow xs_1 = []) \wedge (c_{11} \Rightarrow xs_1 = x_2 : xs_2) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \\
& \wedge (c_{20} \Rightarrow xs_2 = []) \wedge (c_{21} \Rightarrow xs_2 = x_3 : xs_3) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21}) \\
& \wedge (c_{30} \Rightarrow xs_3 = []) \wedge (c_{21} \Rightarrow c_{30})
\end{aligned}
$$

Structured Containers

```
best :: k:Nat -> {xs:[Score] | k <= len xs}
     -> {v:[Score] | k = len v}
```

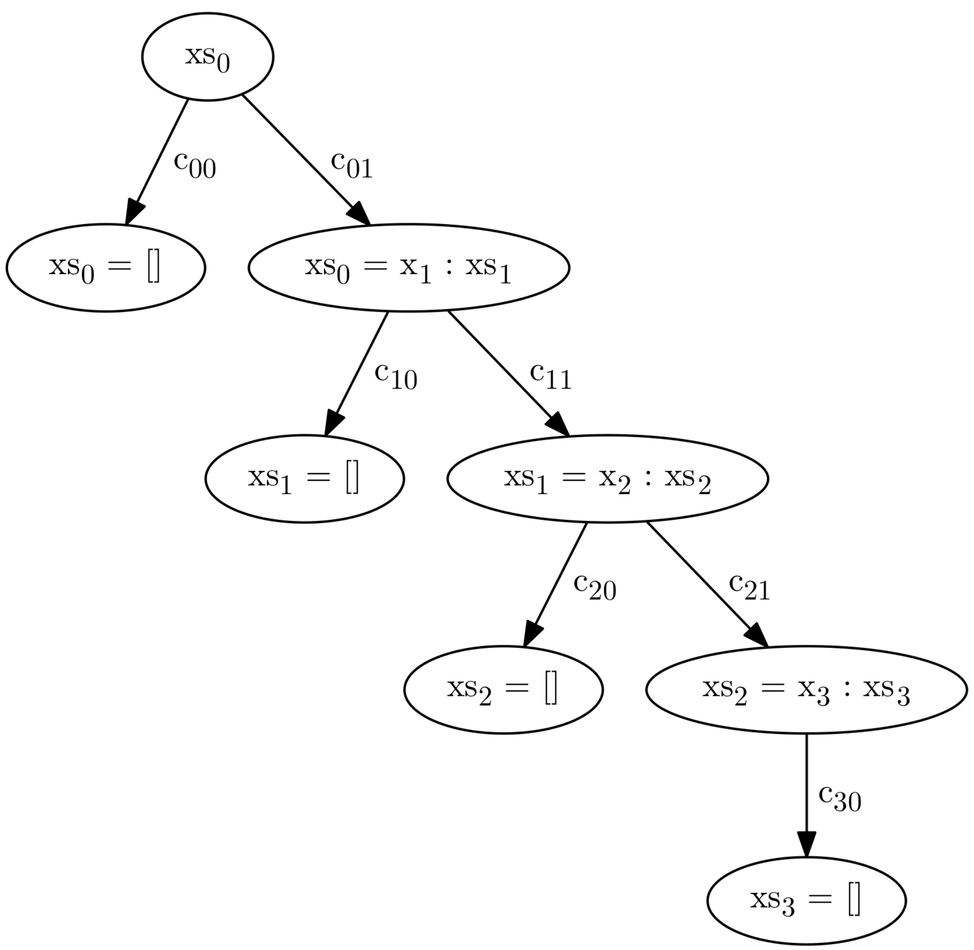


$$
\begin{aligned}
\mathsf{C}_{\mathsf{list}} \doteq\ & (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \wedge (c_{00} \oplus c_{01}) \\
& \wedge (c_{10} \Rightarrow xs_1 = []) \wedge (c_{11} \Rightarrow xs_1 = x_2 : xs_2) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \\
& \wedge (c_{20} \Rightarrow xs_2 = []) \wedge (c_{21} \Rightarrow xs_2 = x_3 : xs_3) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21}) \\
& \wedge (c_{30} \Rightarrow xs_3 = []) \wedge (c_{21} \Rightarrow c_{30})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C}_{\mathsf{size}} \doteq\ & (c_{00} \Rightarrow \mathsf{len}\ xs_0 = 0) \wedge (c_{01} \Rightarrow \mathsf{len}\ xs_0 = 1 + \mathsf{len}\ xs_1) \\
& \wedge (c_{10} \Rightarrow \mathsf{len}\ xs_1 = 0) \wedge (c_{11} \Rightarrow \mathsf{len}\ xs_1 = 1 + \mathsf{len}\ xs_2) \\
& \wedge (c_{20} \Rightarrow \mathsf{len}\ xs_2 = 0) \wedge (c_{21} \Rightarrow \mathsf{len}\ xs_2 = 1 + \mathsf{len}\ xs_3) \\
& \wedge (c_{30} \Rightarrow \mathsf{len}\ xs_3 = 0)
\end{aligned}
$$

Structured Containers

```
best :: k:Nat -> {xs:[Score] | k <= len xs}
     -> {v:[Score] | k = len v}
```



$$
\begin{aligned}
\mathsf{C_{list}} \doteq\ & (c_{00} \Rightarrow xs_0 = []) \wedge (c_{01} \Rightarrow xs_0 = x_1 : xs_1) \wedge (c_{00} \oplus c_{01}) \\
& \wedge (c_{10} \Rightarrow xs_1 = []) \wedge (c_{11} \Rightarrow xs_1 = x_2 : xs_2) \wedge (c_{01} \Rightarrow c_{10} \oplus c_{11}) \\
& \wedge (c_{20} \Rightarrow xs_2 = []) \wedge (c_{21} \Rightarrow xs_2 = x_3 : xs_3) \wedge (c_{11} \Rightarrow c_{20} \oplus c_{21}) \\
& \wedge (c_{30} \Rightarrow xs_3 = []) \wedge (c_{21} \Rightarrow c_{30})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{C_{size}} \doteq\ & (c_{00} \Rightarrow \mathsf{len}\ xs_0 = 0) \wedge (c_{01} \Rightarrow \mathsf{len}\ xs_0 = 1 + \mathsf{len}\ xs_1) \\
& \wedge (c_{10} \Rightarrow \mathsf{len}\ xs_1 = 0) \wedge (c_{11} \Rightarrow \mathsf{len}\ xs_1 = 1 + \mathsf{len}\ xs_2) \\
& \wedge (c_{20} \Rightarrow \mathsf{len}\ xs_2 = 0) \wedge (c_{21} \Rightarrow \mathsf{len}\ xs_2 = 1 + \mathsf{len}\ xs_3) \\
& \wedge (c_{30} \Rightarrow \mathsf{len}\ xs_3 = 0)
\end{aligned}
$$

Enforce relation between `k` and `xs` by adding $k \leq \mathsf{len}\ xs_0$

$$\mathsf{C} \doteq \mathsf{C_{list}} \wedge \mathsf{C_{data}} \wedge \mathsf{C_{size}} \wedge 0 \leq k \leq \mathsf{len}\ xs_0$$

Demo: Targeting `BST`s

Evaluation

Our Claims

1. Target handles highly structured inputs automatically
2. Target generates tests that provide high code coverage

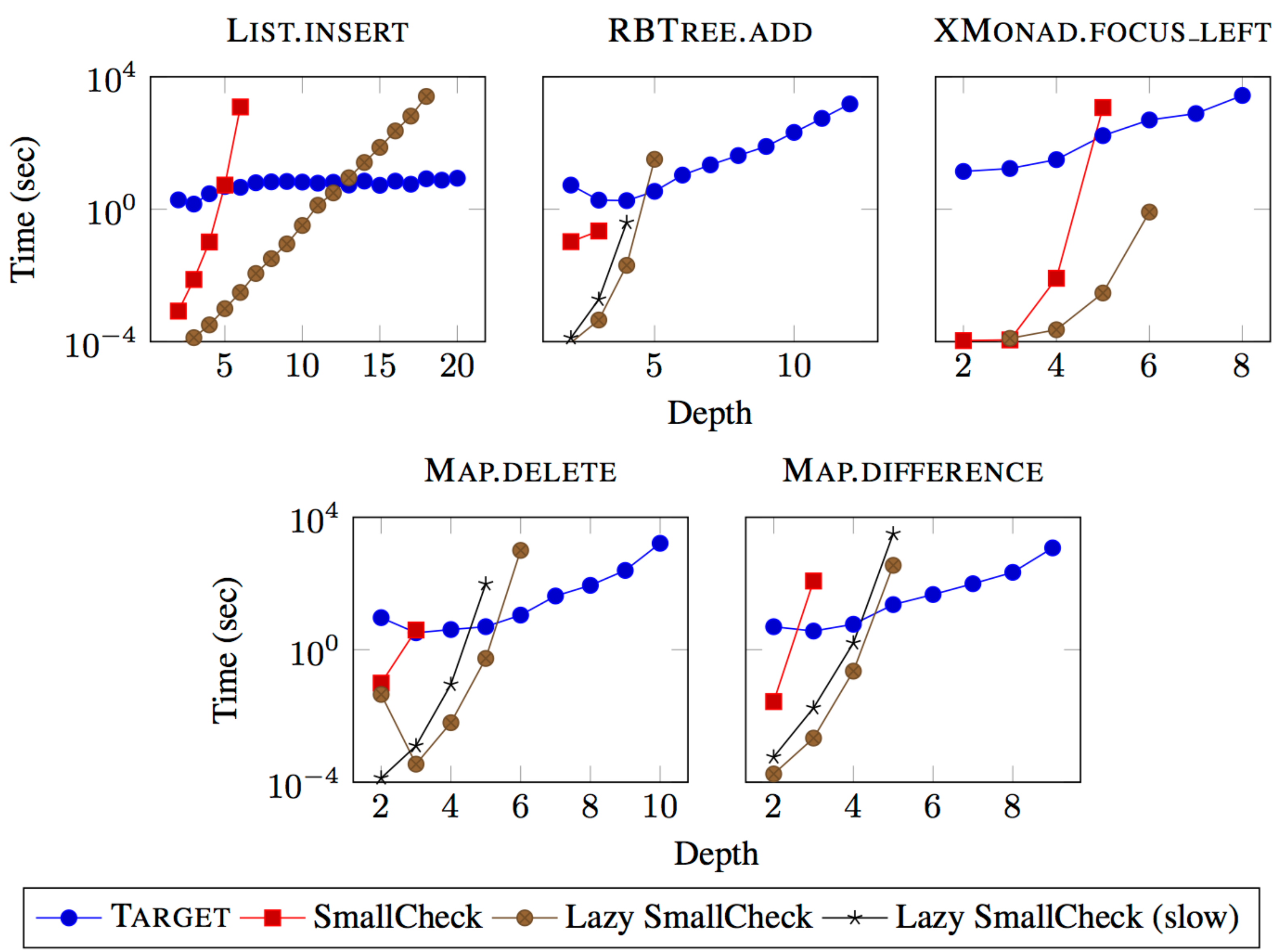
Benchmarks

1. **`Data.Map`**: checked balancing and ordering invariants
2. **`RBTree`**: checked red-black and ordering invariants
3. **`XMonad.StackSet`**: checked uniqueness of windows

Compared Target against QuickCheck and SmallCheck

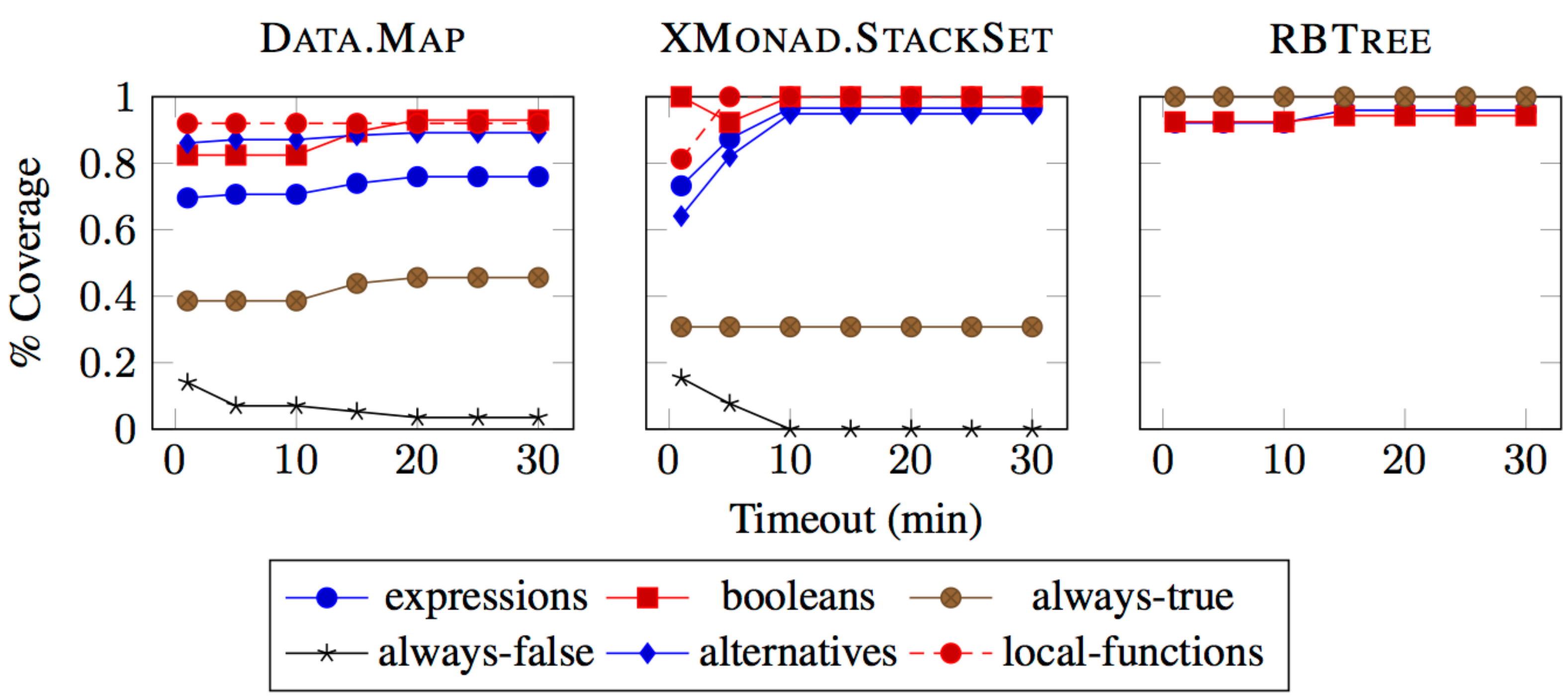
Evaluation: Results

Target checks larger inputs than brute-force



Evaluation: Results

Target provides high coverage with low investment



Takeaway

Target - a new approach for automatically testing functions with preconditions.

- **guarantees** inputs satisfy preconditions
- vs **QuickCheck**: does not require custom generators
- vs **SmallCheck**: defers the onset of input explosion