# Verifying Robustness of Programs Under Structural Perturbations

Clay Thomas and Jacob Bond

December 1, 2017

# Motivation

- An attempt to synthesize the max function using PBE:
  - $(13, 15) \mapsto 15$
  - $(-23, 19) \mapsto 19$
  - $(-75, -13) \mapsto -13$

## Motivation

- An attempt to synthesize the max function using PBE:
  - $(13, 15) \mapsto 15$
  - $(-23, 19) \mapsto 19$
  - $(-75, -13) \mapsto -13$
- Synthesized program: `P(a,b):=return b`

# Motivation

- An attempt to synthesize the max function using PBE:
  - $(13, 15) \mapsto 15$
  - $(-23, 19) \mapsto 19$
  - $(-75, -13) \mapsto -13$
- Synthesized program: `P(a,b):=return b`
- Neither synthesized program, nor synthesizer are *robust*

# Robustness

- Robustness: behaving predictably on uncertain inputs [**?**]

# Robustness

- Robustness: behaving predictably on uncertain inputs [**?**]
- `P(13,15)` $\neq$ `P(15,13)`

# Robustness

- Robustness: behaving predictably on uncertain inputs [**?**]
- `P(13,15)` $\neq$ `P(15,13)`
- - $(15, 13) \mapsto 15$
  - $(-23, 19) \mapsto 19$
  - $(-13, -75) \mapsto -13$

  would synthesize very different program

# Robustness

- Robustness: behaving predictably on uncertain inputs [**?**]
- `P(13,15)` $\neq$ `P(15,13)`
- 
  - $(15, 13) \mapsto 15$
  - $(-23, 19) \mapsto 19$
  - $(-13, -75) \mapsto -13$

  would synthesize very different program
- Synthesize a robust program or develop robust synthesizer

## Robustness Properties

- Continuity: small change to input $\Rightarrow$ small change to output

```
Sort([1,4,3,6])=[1,3,4,6]
Sort([2,3,3,5])=[2,3,3,5]
```

# Robustness Properties

- Continuity: small change to input $\Rightarrow$ small change to output

```
Sort([1,4,3,6])=[1,3,4,6]
Sort([2,3,3,5])=[2,3,3,5]
```

- Permutation: permuting input permutes the output

```
Find([1,4,3,6], 4)=1
```

$$\sigma = (0 \mapsto 2, 1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 0)$$

```
Find([6,3,1,4], 4)=3
```

# Robustness Properties

- Continuity: small change to input $\Rightarrow$ small change to output

```
Sort([1,4,3,6])=[1,3,4,6]
Sort([2,3,3,5])=[2,3,3,5]
```

- Permutation: permuting input permutes the output

```
Find([1,4,3,6], 4)=1
```

$$\sigma = (0 \mapsto 2, 1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 0)$$

```
Find([6,3,1,4], 4)=3
```

- Permutation: permuting input leaves output invariant

```
Sort([1,4,3,6])=[1,3,4,6]
Sort([6,3,1,4])=[1,3,4,6]
```

# Verifying Continuity

- Consider

```
if x ≥ 0 then
    r := y
else
    r := z
```

# Verifying Continuity

- Consider

```
if x ≥ 0 then
    r := y
else
    r := z
```

- If $y \neq z$, discontinuous at $x = 0$

# Verifying Continuity

- Consider

```
if x ≥ 0 then
    r := y
else
    r := z
```

- If $y \neq z$, discontinuous at $x = 0$
- Proof rule:

$$\frac{c \vdash \mathrm{Cont}(P_1, \mathrm{In}, \mathrm{Out}) \qquad c \vdash \mathrm{Cont}(P_2, \mathrm{In}, \mathrm{Out}) \qquad c' \vdash \mathrm{Cont}(b, \mathrm{Var}(b)) \qquad (c \wedge \neg c') \vdash \mathrm{Out}_{P_1} = \mathrm{Out}_{P_2}}{c \vdash \mathrm{Cont}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, \mathrm{In}, \mathrm{Out})}$$

# Verifying Continuity

- Consider

  **if** $x \geq 0$ **then**
  $\quad r := y$
  **else**
  $\quad r := z$

- If $y \neq z$, discontinuous at $x = 0$
- Proof rule:

$$\frac{c \vdash \mathrm{Cont}(P_1, \mathrm{In}, \mathrm{Out}) \qquad c \vdash \mathrm{Cont}(P_2, \mathrm{In}, \mathrm{Out}) \qquad c' \vdash \mathrm{Cont}(b, \mathrm{Var}(b)) \qquad (c \wedge \neg c') \vdash \mathrm{Out}_{P_1} = \mathrm{Out}_{P_2}}{c \vdash \mathrm{Cont}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, \mathrm{In}, \mathrm{Out})}$$

- Only applicable to numerical perturbations

## If-Elsif-Elsif-Else Cost

- 

$$\frac{\begin{array}{cc} c \vdash \mathrm{Cont}(P_1, \mathrm{In}, \mathrm{Out}) & c \vdash \mathrm{Cont}(P_2, \mathrm{In}, \mathrm{Out}) \\ c' \vdash \mathrm{Cont}(b, \mathrm{Var}(b)) & (c \wedge \neg c') \vdash \mathrm{Out}_{P_1} = \mathrm{Out}_{P_2} \end{array}}{c \vdash \mathrm{Cont}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, \mathrm{In}, \mathrm{Out})}$$

# If-Elsif-Elsif-Else Cost

- $$\frac{c \vdash \mathrm{Cont}(P_1, \mathrm{In}, \mathrm{Out}) \qquad c \vdash \mathrm{Cont}(P_2, \mathrm{In}, \mathrm{Out}) \qquad c' \vdash \mathrm{Cont}(b, \mathrm{Var}(b)) \qquad (c \wedge \neg c') \vdash \mathrm{Out}_{P_1} = \mathrm{Out}_{P_2}}{c \vdash \mathrm{Cont}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, \mathrm{In}, \mathrm{Out})}$$
- Given `if-elsif-elsif-else`,

# If-Elsif-Elsif-Else Cost

- $$\frac{\begin{array}{cc} c \vdash \mathrm{Cont}(P_1, \mathrm{In}, \mathrm{Out}) & c \vdash \mathrm{Cont}(P_2, \mathrm{In}, \mathrm{Out}) \\ c' \vdash \mathrm{Cont}(b, \mathrm{Var}(b)) & (c \wedge \neg c') \vdash \mathrm{Out}_{P_1} = \mathrm{Out}_{P_2} \end{array}}{c \vdash \mathrm{Cont}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, \mathrm{In}, \mathrm{Out})}$$

- Given `if-elsif-elsif-else`,
  - Reason about each branch (4 branches)

# If-Elsif-Elsif-Else Cost

- $$\frac{\begin{array}{cc} c \vdash \mathrm{Cont}(P_1, \mathrm{In}, \mathrm{Out}) & c \vdash \mathrm{Cont}(P_2, \mathrm{In}, \mathrm{Out}) \\ c' \vdash \mathrm{Cont}(b, \mathrm{Var}(b)) & (c \wedge \neg c') \vdash \mathrm{Out}_{P_1} = \mathrm{Out}_{P_2} \end{array}}{c \vdash \mathrm{Cont}(\textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2, \mathrm{In}, \mathrm{Out})}$$

- Given `if-elsif-elsif-else`,
  - Reason about each branch (4 branches)
  - Check equivalence (3 comparisons)

# Cartesian Hoare Logic

- Robustness requires 2 executions

# Cartesian Hoare Logic

- Robustness requires 2 executions
- Verified using product program

# Cartesian Hoare Logic

- Robustness requires 2 executions
- Verified using product program
  - $P_1 \circledast P_2$ is simultaneous execution

# Cartesian Hoare Logic

- Robustness requires 2 executions
- Verified using product program
  - $P_1 \circledast P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs

# Cartesian Hoare Logic

- Robustness requires 2 executions
- Verified using product program
  - $P_1 \circledast P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:

# Cartesian Hoare Logic

- Robustness requires 2 executions
- Verified using product program
  - $P_1 \circledast P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:
  - Determinism:

$$\|\vec{x_1} = \vec{x_2}\| f(\vec{x}) \| ret_1 = ret_2 \|$$

## Cartesian Hoare Logic

- Robustness requires 2 executions
- Verified using product program
  - $P_1 \circledast P_2$ is simultaneous execution
- Cartesian Hoare Logic reasons about product programs
- Cartesian Hoare Triple examples:
  - Determinism:

$$\|\vec{x_1} = \vec{x_2}\| f(\vec{x}) \| ret_1 = ret_2\|$$

  - Symmetry:

$$\|x_1 = y_2 \wedge x_2 = y_1\| f(x, y) \| ret_1 = ret_2\|$$

# If-Elsif-Elsif-Else Cost

- Given `if-elsif-elsif-else`, must reason about product of each pair of branches (16 pairs):

$$
\begin{gathered}
P_1 \circledast P_1 \\
\vdots \\
P_1 \circledast P_4 \\
P_2 \circledast P_1 \\
\vdots \\
P_4 \circledast P_4
\end{gathered}
$$

## Motivation

- Sanity checks and bug finding

- Functions invariant under order of list
    - max, sum, sort

- Data structures with representing something else
- Invariance under value it represents
    - binary search trees, heaps, hash sets

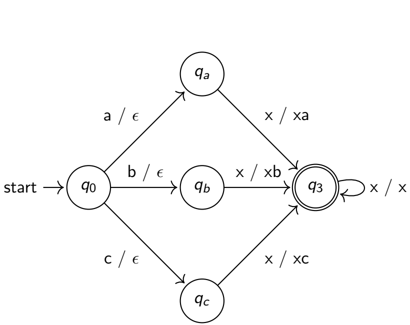
## Lists – Invariance under order

Given an array $a$

- Let $a_{swap}$ be $a$ with its first and second entry swapped
  - $[a[1], a[0], a[2], a[3], \ldots, a[n]]$
- Let $a_{rot}$ be $a$ rotated by 1
  - $[a[1], a[2], a[3], \ldots a[n], a[0]]$

Lemma: If for any $a$, $P(a) = P(a_{swap}) = P(a_{rot})$, then for any permutation $a'$ of $a$, we have $P(a) = P(a')$.
Proof: Math

# Automata – Invariance under order



# Automata – Invariance under order

Theorem: Given a deterministic automata $M$, we can check if $M$ is invariant under the order of its input in time $O(|\Sigma|^2 |M| \log(|\Sigma||M|))$.

Proof:

- Construct the machines $M_{swap}$ and $M_{rot}$ by composing $M$ with those machines
  - Requires $O(|\Sigma||M|)$ states
- Check if $L(M_{swap}) = L(M) = L(M_{rot})$
  - E.g. by state minimization in time $O(n|\Sigma| \log n)$

## Programs – Invariance under order

- maxList([x]) $=$ x
- maxList([x, ...xs...]) $=$ max(x, maxList(xs))

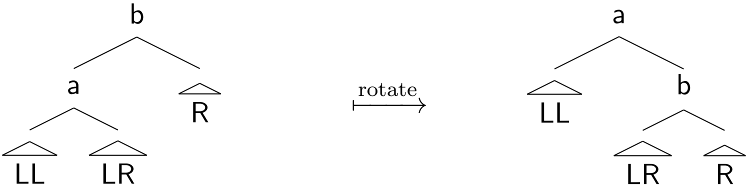
- Verifying maxList$(a)$ $=$ maxList$(a_{swap})$ has one case:

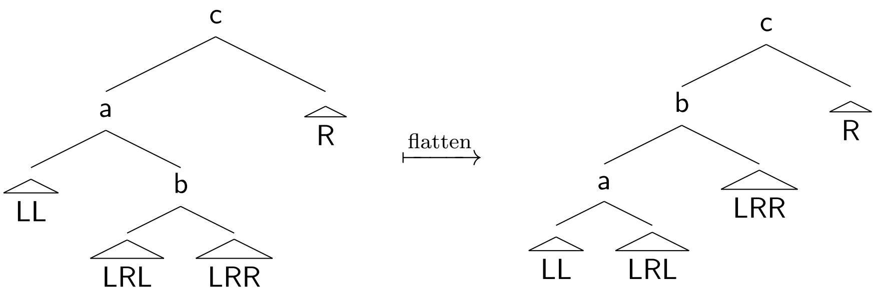
$$
\begin{array}{rcl}
maxList([x, y, ...xs...]) & \stackrel{?}{=} & maxList([y, x, ...xs...]) \\
|| & & || \\
max(x, maxList([y, ...xs...])) & & max(y, maxList([x, ...xs...])) \\
|| & & || \\
max(x, max(y, maxList(xs))) & & max(y, max(x, maxList(xs))) \\
|| & & || \\
max(x, max(y, z)) & & max(y, max(x, z))
\end{array}
$$

# Binary Search Trees

- For lists, two simple permutations generated all permutations
- Goal: similar permutations for BSTs

# Binary Search Trees



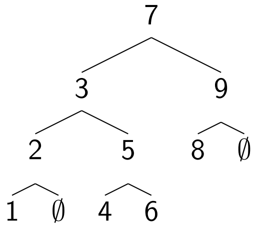
# Binary Search Trees

It suffices to show

- Every tree can be transformed into a “normal form” (i.e. list)
- Every operation is reversable

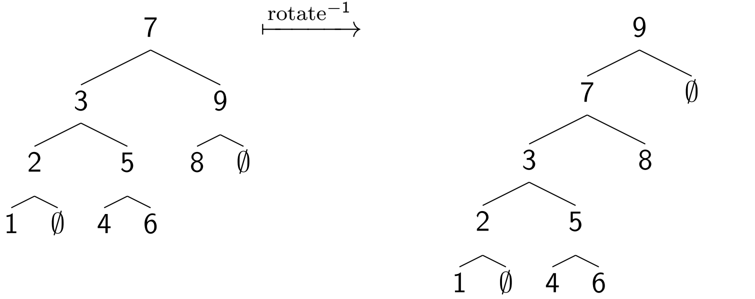
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



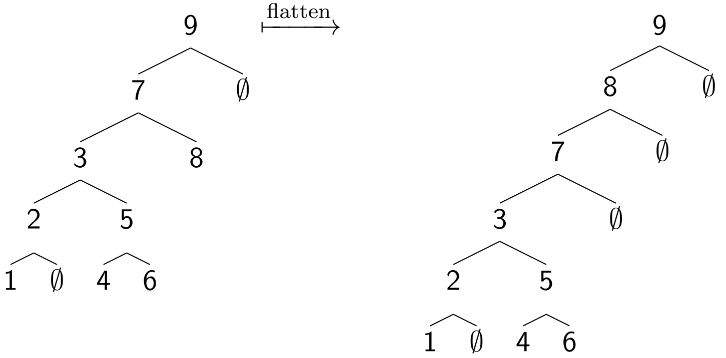
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



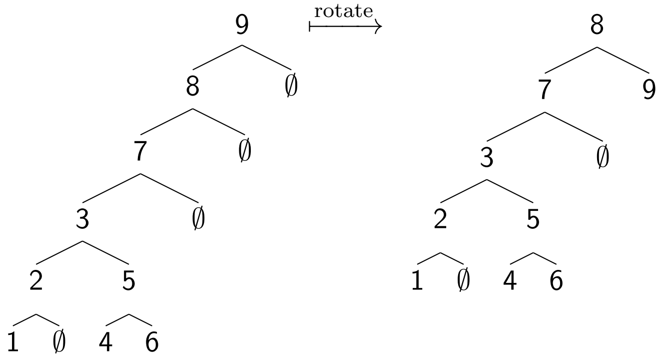
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



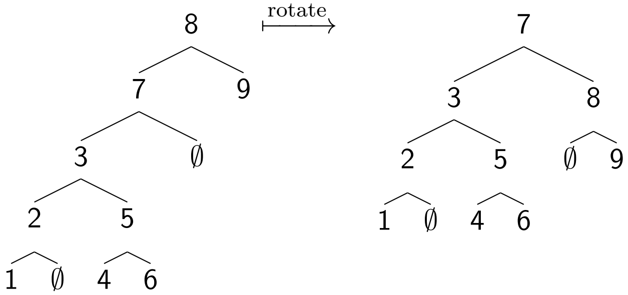
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



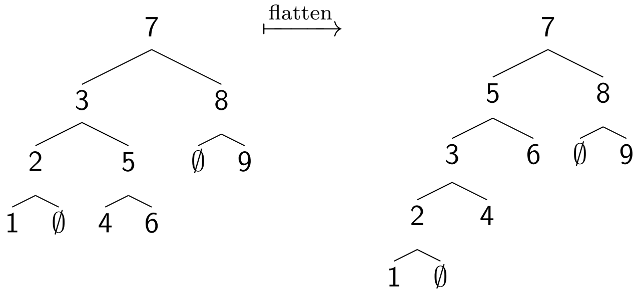
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



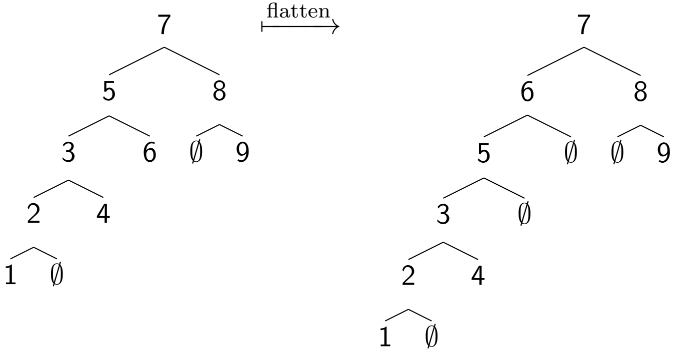
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



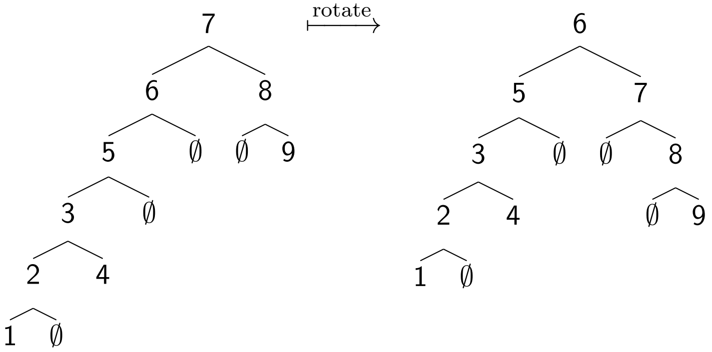
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



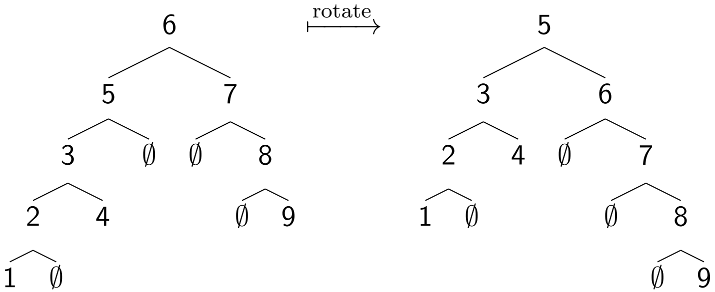
# Binary Search Trees – Proof by example

- Every tree can be transformed into a "normal form" (i.e. list)



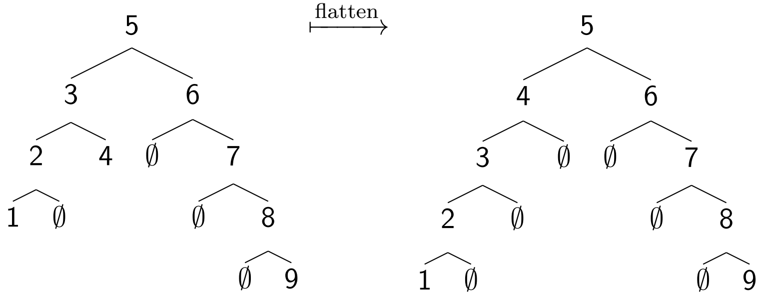
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



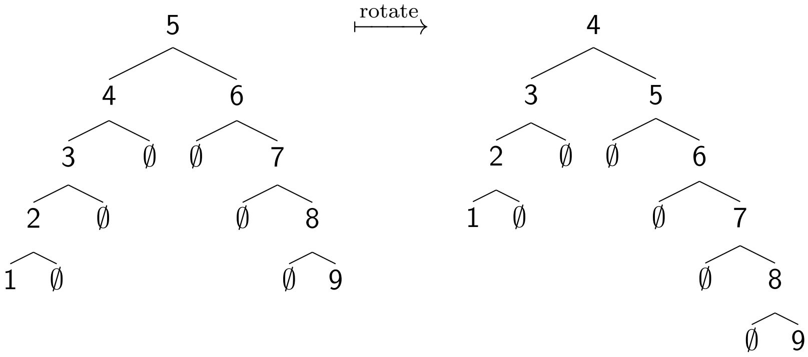
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



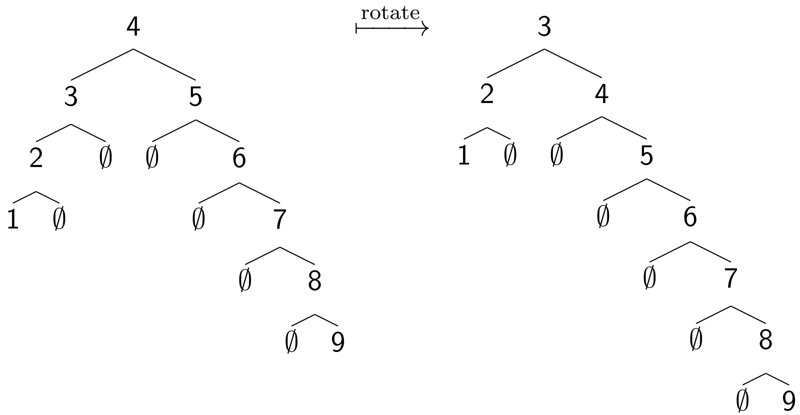
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



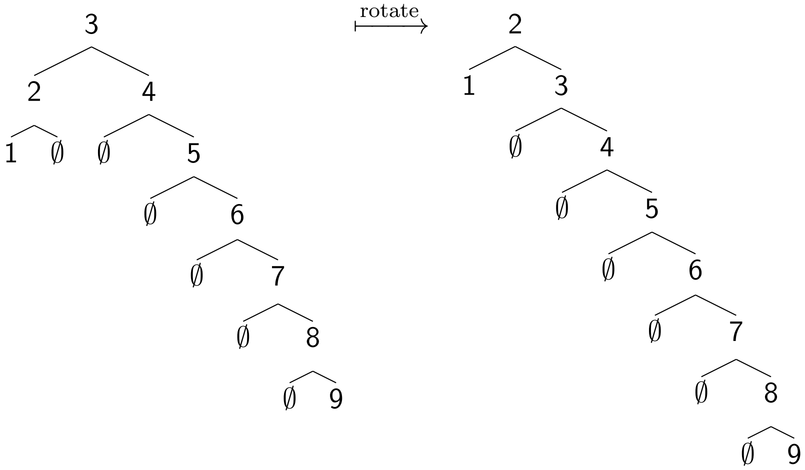
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



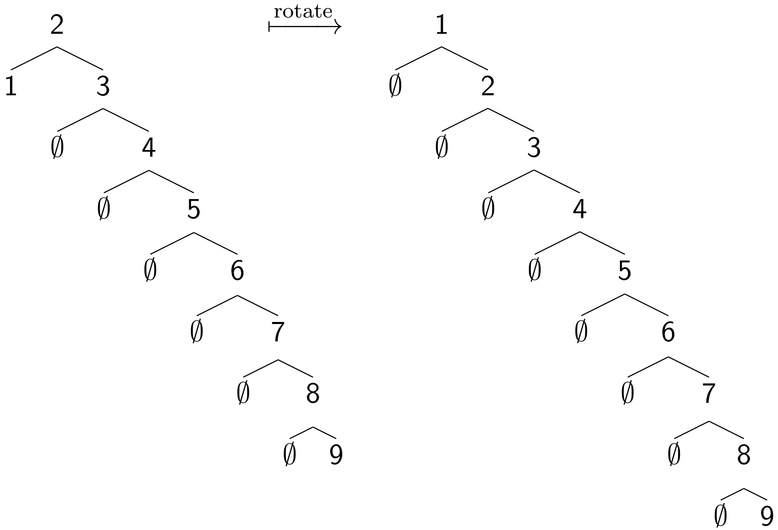
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



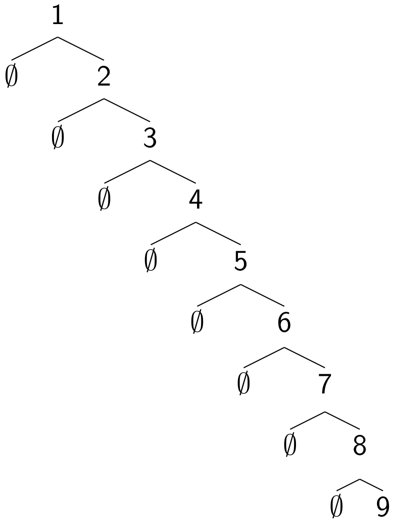
# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



# Binary Search Trees – Proof by example

- Every tree can be transformed into a “normal form” (i.e. list)



## More General Procedure

Invariance of a program $P : T \to Z$ relative to a function $f : T \to T'$

- E.g. $f : BST \to List$

Observation: The following are equivalent:

- $f(x) = f(y) \implies P(x) = P(y)$
- There exists a program $\widetilde{P} : T' \to Z$ such that $P = \widetilde{P} \circ f$

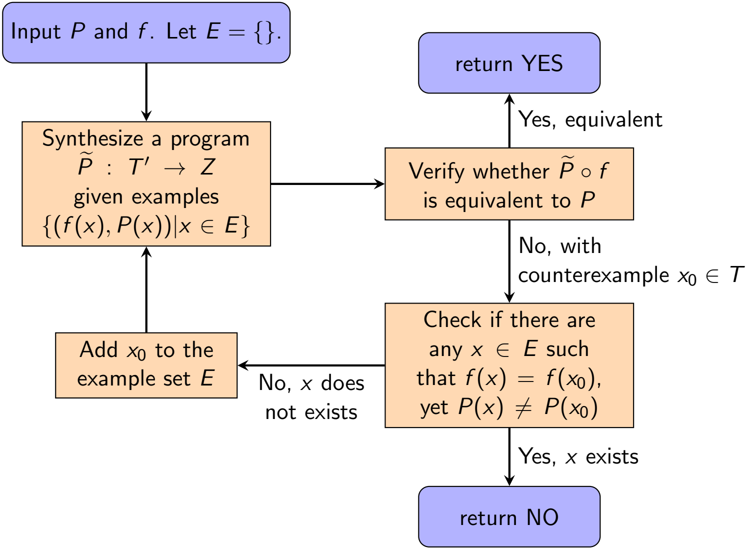
$$\begin{tikzcd} T \arrow[rr, "P"] \arrow[dr, "f"'] & & Z \\ & T' \arrow[ur, dashed, "\widetilde{P}"'] & \end{tikzcd}$$

# More General Procedure

- Idea: Synthesize a witness to the invariance
  - A function $\widetilde{P} : T' \to Z$
- $P$ and $f$ provide a *full specification* of $\widetilde{P}$
- Counterexample guided inductive synthesis

$$\begin{tikzcd} T \arrow[rr, "P"] \arrow[dr, "f"] & & Z \\ & T' \arrow[ur, dashed, "\widetilde{P}"] & \end{tikzcd}$$

# More General Procedure



# Future Directions

- Develop proof rules for discrete perturbations

# Future Directions

- Develop proof rules for discrete perturbations
- Address branching in 2-safety properties

## Future Directions

- Develop proof rules for discrete perturbations
- Address branching in 2-safety properties
-