# 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)`
- would synthesize very different program
  - $(15, 13) \mapsto 15$
  - $(19, -23) \mapsto 19$
  - $(-13, -75) \mapsto -13$

# Robustness

- Robustness: behaving predictably on uncertain inputs [**?**]
- `P(13,15)` $\neq$ `P(15,13)`
- would synthesize very different program
  - $(15, 13) \mapsto 15$
  - $(19, -23) \mapsto 19$
  - $(-13, -75) \mapsto -13$
- 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 leaves output invariant

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

# 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 leaves output invariant

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

- **Simultaneous Permutation**: permuting all inputs leaves output invariant (`Grade(responses,answers)`)

```
Grade([sqrt(x^2), 1/e, 6.5], [abs(x), e^-1, 13/2])=1
```

rearrange problem parts

```
Grade([1/e, 6.5, sqrt(x^2)], [e^-1, 13/2, abs(x)])=1
```

# Verifying Continuity

- Consider

1: **if** $x \geq 0$ **then**
2: $\quad r := y$
3: **else**
4: $\quad r := z$

# Verifying Continuity

- Consider

  1: **if** $x \geq 0$ **then**
  2: $\quad r := y$
  3: **else**
  4: $\quad r := z$

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

# Verifying Continuity

- Consider

```
1: if x ≥ 0 then
2:     r := y
3: else
4:     r := z
```

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

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

# Verifying Continuity

- Consider

```
1: if x ≥ 0 then
2:     r := y
3: else
4:     r := z
```

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

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

- Only applicable to numerical perturbations

# 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:

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

# 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:

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

- Requires specifying property in first-order logic

# Our Contributions

- Small sets of perturbations
- Small sets of perturbations

- Sanity checks and bug finding

# 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

# Programs – Invariance under order

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

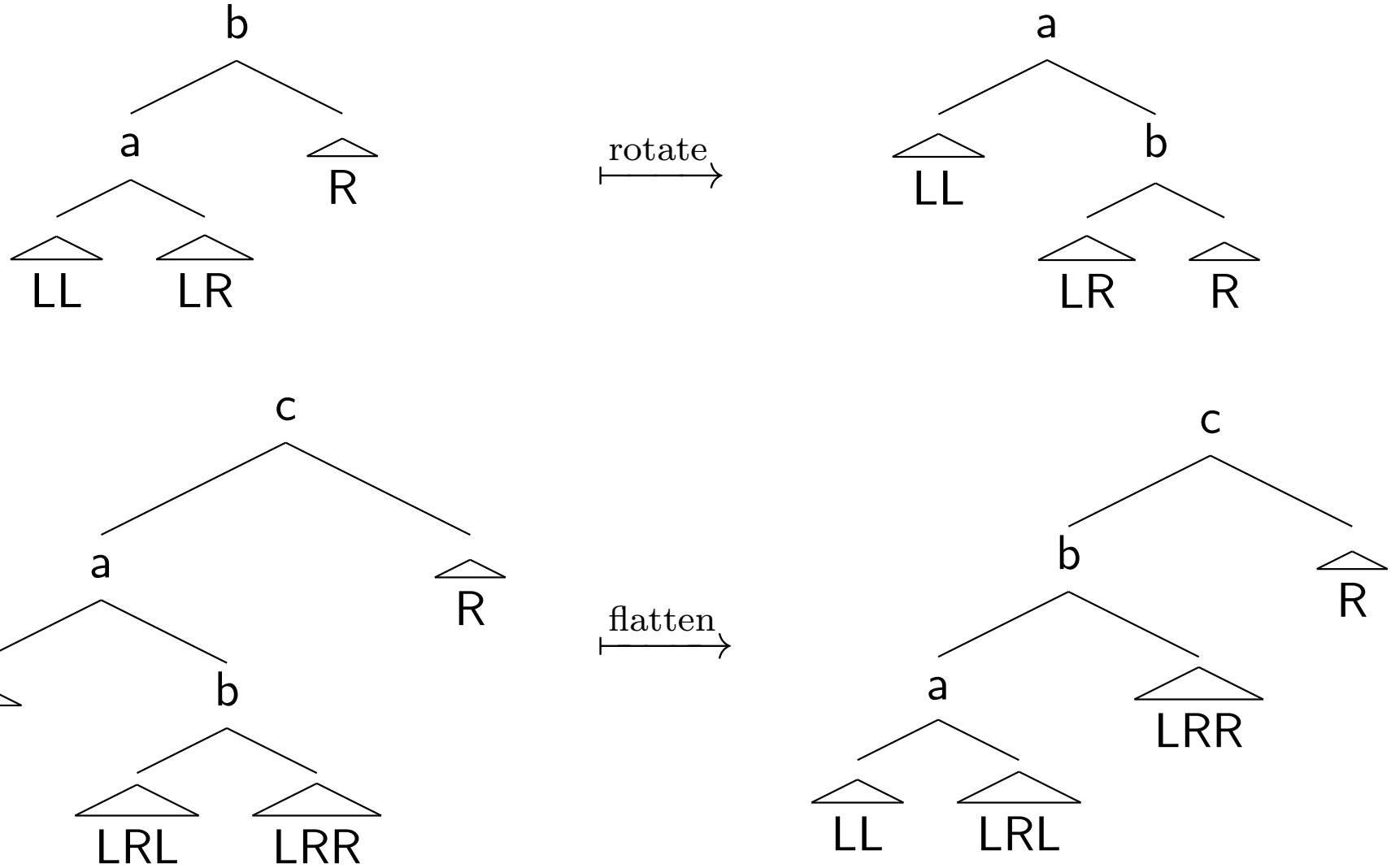
- Verifying maxList($a$) = maxList($a_{swap}$) has one case:

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

# Binary Search Trees

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

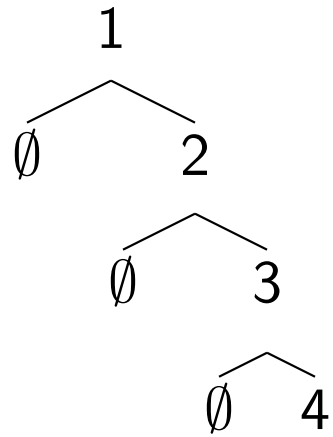
# Binary Search Trees



# Binary Search Trees

It suffices to show

- Every tree can be transformed into a "normal form" (i.e. list)
  - "flatten" straightens out the tree
  - "rotate" lets you straighten all the parts
- Every operation is reversable



# Lists and Binary Search Trees

- Can check robustness under ALL permutations by checking just TWO permutations

# More General Procedure

- Sets of permutations are case-by-case
- Goal: formulation of invariance
  - Useful
  - Easy to code/express
  - Checkable

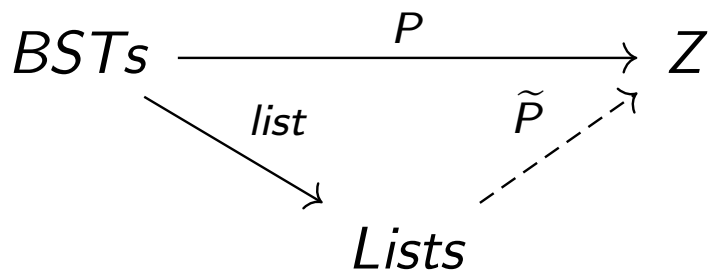
# More General Procedure

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

- $f(t)$ gives a "canonical representative" of $t$
- For concreteness, $f = \mathit{list} : \mathit{BST} \to \mathit{List}$

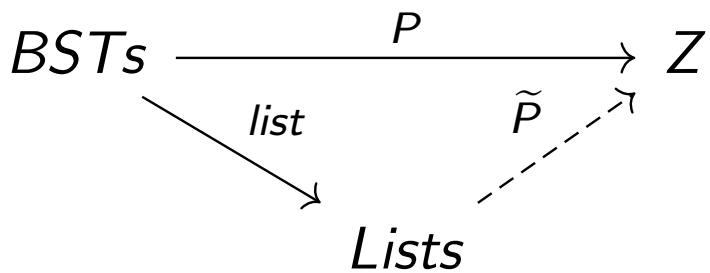
Observation: The following are equivalent:

- $\mathit{list}(x) = \mathit{list}(y) \implies P(x) = P(y)$
- There exists a program $\widetilde{P} : \mathit{Lists} \to Z$ such that $P(t) = \widetilde{P}(\mathit{list}(t))$

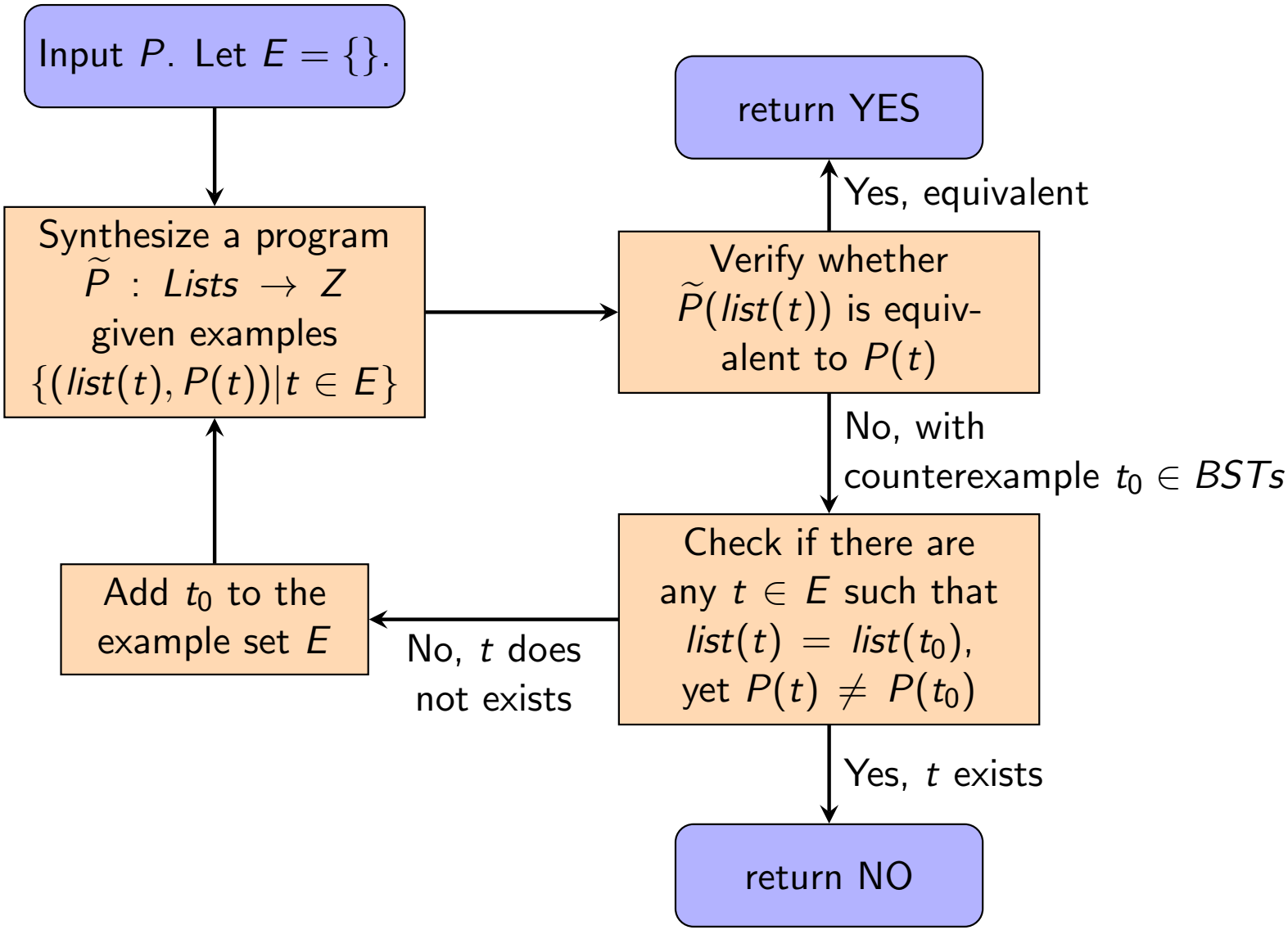


# More General Procedure

- Idea: Synthesize a witness to the invariance
  - A function $\widetilde{P} : \mathit{Lists} \to Z$
- $P$ and *list* provide a *full specification* of $\widetilde{P}$
- Counterexample guided inductive synthesis



# 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
-

# Future Directions

- Develop proof rules for discrete perturbations
- Address branching in 2-safety properties
-
- Find more data structures with small perturbation sets

# Future Directions

- Develop proof rules for discrete perturbations
- Address branching in 2-safety properties
-
- Find more data structures with small perturbation sets
- Speed up our general procedure

# Future Directions

- Develop proof rules for discrete perturbations
- Address branching in 2-safety properties
- 
- Find more data structures with small perturbation sets
- Speed up our general procedure
- Implement!