

# Programs You Can Trust

Edwin Westbrook

Advisor: Aaron Stump

# Motivation

---

- Programs are ubiquitous (e.g. cell phones, cars, subways, airplanes)
- Incorrect programs are bad
  - Cost lots of time and money
  - Can even cause death (e.g. programs in airplanes)
- Writing correct programs is hard
  - Even checking programs is hard
- How can we trust our programs?

# General Approach

---

- Program Verification
  - Requires a specification of “correct program”
  - Requires guarantee that program meets specification
- The types-based approach to verification
  - Program type is specification
  - Type-checking is guarantee
- Existing strongly-typed languages (e.g. Java, ML) allow some specification
  - Cannot ensure more complicated data structure properties
- Solution: indexed (or dependent) types

# Outline

---

- Motivating example: binary search tree insertion
- Safe BSTs
- Verified insert with safe BSTs
- Mutable State
  - This is our original research

## Note about syntax

---

- Warning: the syntax you are about to see is faked.
- We will use Java syntax for expository purposes in this talk.
- The material in this talk is actually implemented in a different language:
  - Implemented as the language RSP1
  - see Westbrook, Stump, Wehrman, ICFP '05

# Motivating Example

---

```
void insert (Tree T, int x) {  
    if (x ≤ T.data)  
        if (T.left == empty)  
            T.left  
                = new Tree (x, empty, empty);  
        else  
            insert (T.left, x);  
    else  
        ... symmetrical case  
}
```

# Motivating Example

---

```
void insert (Tree T, int x) {  
    if (x ≤ T.data)  
        if (T.left == empty)  
            T.data  
                = new Tree (x, empty, empty);  
        else  
            insert (T.left, x);  
    else  
        ... symmetrical case  
}
```

# Motivating Example

---

```
void insert (Tree T, int x) {  
    if (x ≤ T.data)  
        if (T.left == empty)  
            T.data      Java catches datatype bugs  
                = new Tree (x, empty, empty);  
        else  
            insert (T.left, x);  
        else  
            ...symmetrical case  
}
```

# Motivating Example

---

```
void insert (Tree T, int x) {  
    if (x ≤ T.data)  
        if (T.left == empty)  
            T.right  
                = new Tree (x, empty, empty);  
        else  
            insert (T.left, x);  
    else  
        ... symmetrical case  
}
```

# Motivating Example

---

```
void insert (Tree T, int x) {  
    if (x ≤ T.data)  
        if (T.left == empty)  
            T.right      Java does not catch BST bugs  
                = new Tree (x, empty, empty);  
        else  
            insert (T.left, x);  
        else  
            ... symmetrical case  
}
```

- This insert can create invalid BSTs!

# Trusting insert

---

- Type for insert:

```
void insert(Tree T, int x)
```

- only ensures it returns a Tree

- We really want:

```
void insert(BST T, int x)
```

- Ensures insert only constructs BSTs

- Requires a “safe” BST type such that:

- We can make any valid BST

- We cannot make invalid BSTs

# Safe BSTs?

---

- Safe BST is like Tree:

```
class BST {  
    int data;  
    BST left, right;  
}
```

**except:**

- $\text{data} \geq n$  for any  $n$  in left
  - $\text{data} \leq n$  for any  $n$  in right
- Cannot represent these guarantees in normal Java!

# Safe BSTs

---

- If we **index** the BST type, we can get:

```
class BST<int m,n> { // BST covering range m-n
    int data;
    BST<m,data> left;
    BST<data,n> right;
    LEQ<m,data> g1; // guarantee that  $m \leq \text{data}$ 
    LEQ<data,n> g2; // guarantee that  $\text{data} \leq n$ 
}
```

- Now we need to represent guarantees

# Guarantees

---

- What is a “guarantee”?
  - Witness to some property being true
  - Should only be able to make true guarantees
- For  $\text{LEQ}\langle m, n \rangle$  to guarantee  $m \leq n$ , need:
  - Can make an  $\text{LEQ}\langle m, n \rangle$  whenever  $m \leq n$
  - Cannot if  $m \not\leq n$

# Defining LEQ

---

- Rules that define  $\leq$ :
  - $n \leq n$ , for any  $n$
  - If  $n \leq m$ , then  $n \leq m+1$

# Defining LEQ

---

- Rules that define  $\leq$ :
  - $n \leq n$ , for any  $n$
  - If  $n \leq m$ , then  $n \leq m+1$

- These become:

```
interface LEQ<int m,n>;
```

```
class LEQ-eq<int n> implements LEQ<n,n> {};
```

```
class LEQ-less<int m,n> implements LEQ<m,n> {  
    LEQ<m,n-1> g; // Cannot be null  
};
```



# insert Revisited

---

```
void insert (BST<m,n> T, int x,  
            LEQ<m,x> g-m-x, LEQ<x,n> g-x-n) {  
    if (x ≤ T.data) {  
        LEQ<x,T.data> g-x-data; // comes from the if  
  
    else ...}
```

# insert Revisited

---

```
void insert (BST<m,n> T, int x,  
            LEQ<m,x> g-m-x, LEQ<x,n> g-x-n) {  
  if (x ≤ T.data) {  
    LEQ<x,T.data> g-x-data; // comes from the if  
    if (T.left == empty)  
      T.left // This is a new BST<m,T.data>  
            = new Tree (x, empty, empty, g-m-x, g-x-data);  
  
  else ...}
```

# insert Revisited

---

```
void insert (BST<m,n> T, int x,  
            LEQ<m,x> g-m-x, LEQ<x,n> g-x-n) {  
    if (x ≤ T.data) {  
        LEQ<x,T.data> g-x-data; // comes from the if  
        if (T.left == empty)  
            T.left // This is a new BST<m,T.data>  
                = new Tree (x, empty, empty, g-m-x, g-x-data);  
        else  
            insert (T.left, x, g-m-x, g-x-data);  
            // T.left is already a BST<m,T.data>  
    } else ... }
```

# Mutable State

---

- But there's one small problem
- Consider:  
    `int x = 1, y = 2;`  
    `LEQ<x,y> g = ...`  
    `y = 0;`
- `g` now guarantees  $1 \leq 0$ !
- The problem: cannot let mutable values index types

## Safe BSTs (again)

---

- data in BST must be read-only:

```
class BST<int m,n> { // BST covering range m-n
    const int data;
    BST<m,data> left;
    BST<data,n> right;
    LEQ<m,data> g1; // guarantee that m ≤ data
    LEQ<data,n> g2; // guarantee that data ≤ n
}
```

# Mutable State

---

- Values of indexed type *can* be mutated
  - E.g. can still modify `T.left` in `insert`
- See: Westbrook, Stump, Wehrman. “A Language-Based Approach to Functionally Correct Imperative Programming.” ICFP '05
  - First work to combine mutable state with indexed types

# Conclusion

---

- Indexed types can:
  - Represent guarantees about data
  - Help verify code (even with mutable data structures)
- Future work:
  - Verified red-black trees (Garrett Eardly)
  - Limits of indexed types for specifying data structure properties
- Acknowledgements:
  - Aaron Stump (my advisor)
  - CL Group (especially Ian Wehrman)
  - Everyone who has helped with this talk