An Example Formalisation

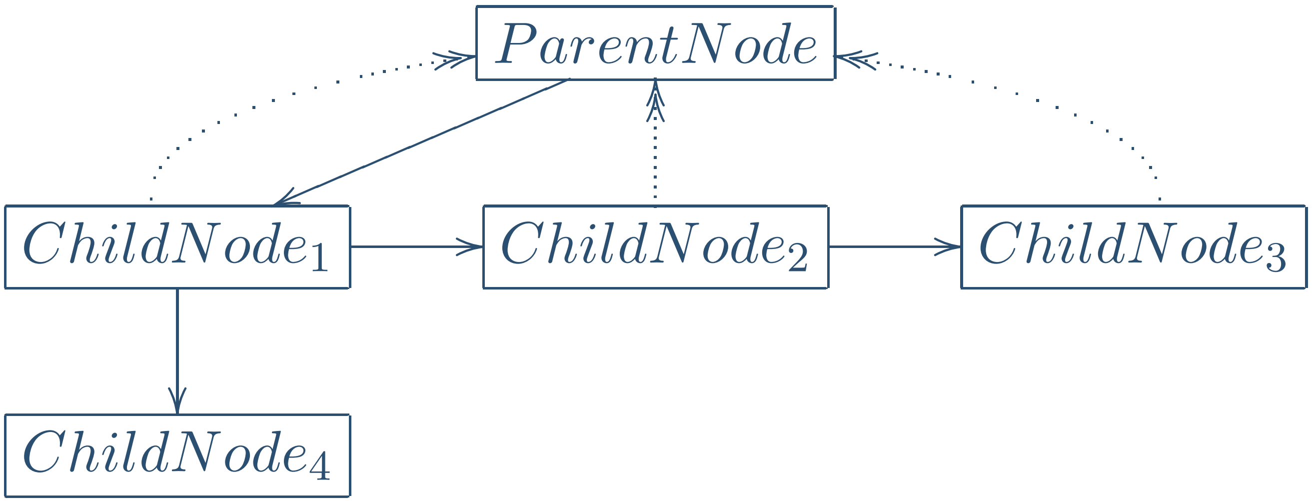
Lars-Åke Fredlund

2010–2011

Task

- Someone has implemented a Java class for Linked Trees (in reality 80 implementations!)

- Link structure:



- Tree nodes or **Positions** store data, and are used to traverse the tree structure

Tree Class Skeleton

```
public class LTree<E> {
    // Add root node
    public Position<E> addRoot(E e) { ... }

    // Returns root, parent, and children nodes
    public Position<E> root() ... { ... }
    public Position<E> parent(Position<E> v) ... { ... }
    public Iterable<Position<E>> children() { ... }

    // Modifies content of a node
    public E replace(Position<E> v, E e) { ... }

    // Add an element e as a child of the node at positio
    public Position<E> addChild(E e,Position<E> v) { ...

    // Add a subtree t as a child of the node at position
    public Position<E> addChild(Tree<E> t,Position<E> v)
}
```

Checking the Implementation

- How to check that the implementation is correct?

Checking the Implementation

- How to check that the implementation is correct?
- Normal answer: inspect code by hand

Checking the Implementation

- How to check that the implementation is correct?
- Normal answer: inspect code by hand
- **But not very reliable. And checking 80 implementations by hand???**

Checking the Implementation

- How to check that the implementation is correct?
- Normal answer: inspect code by hand
- **But not very reliable. And checking 80 implementations by hand???**
- Other normal answer: we write tests

A Typical Test Case

```
LTree<Integer> tree = new LTree<Integer>();

if (!tree.isEmpty()) {
  System.out.println("*** Error: tree should be empty");
  throw new RuntimeException();
}

try {
  tree.root();
  System.out.println("*** Error: tree.root() on an empty
  throw new RuntimeException();
} catch (EmptyTreeException e) { };

System.out.println("all ok");
```

Another Test Case

```
tree.addChild(1, root);
tree.addChild(2, root);
tree.addChild(3, root);

// Count the number of nodes through the Iterable
int numNodes=0;
Iterable<Position<Integer>> iter = tree.children();
for (Position<Integer> p: iter) ++numNodes;
if (numNodes !=4)
  System.out.println("*** Error: root node does not have

System.out.println("all ok");
```

Writing Test Cases

- This gets boring quickly...and takes a **lot of time**
- How can we be sure that we have written the right tests?
- **Or:** How can we be sure that we have run enough tests?

Writing Test Cases

- This gets boring quickly...and takes a **lot of time**
- How can we be sure that we have written the right tests?
- **Or:** How can we be sure that we have run enough tests?
- Traditional measures: how large percentage of the program lines of the program have been touched by any tests
- **Or:** how large percentage of the paths through the program have been touched by any test

Writing Test Cases

- This gets boring quickly...and takes a **lot of time**
- How can we be sure that we have written the right tests?
- **Or:** How can we be sure that we have run enough tests?
- Traditional measures: how large percentage of the program lines of the program have been touched by any tests
- **Or:** how large percentage of the paths through the program have been touched by any test
- **But:** it is well known that these test measures are dangerous.
- A program can have been tested under a good test coverage and contain horrible bugs

Lets Start Thinking About Writing Models

- We want to have an alternative, **correct**, implementation of the LTree class (`LTreeGood`)

Lets Start Thinking About Writing Models

- We want to have an alternative, **correct**, implementation of the LTree class (`LTreeGood`)
- Moreover we want to **automatically** generate a large set of test sequences:

```
Seq1: T=new LTree(); root=T.addRoot(2); ...; T.replace(root,4); ..
Seq2: T=new LTree(); root=T.addRoot(8); ...; addChild(5,root); ...
...
Seq1000: ...
```

Lets Start Thinking About Writing Models

- We want to have an alternative, **correct**, implementation of the LTree class (`LTreeGood`)
- Moreover we want to **automatically** generate a large set of test sequences:

```
Seq1: T=new LTree(); root=T.addRoot(2); ...; T.replace(root,4); ..
Seq2: T=new LTree(); root=T.addRoot(8); ...; addChild(5,root); ...
...
Seq1000: ...
```

- And we want to make sure that **for every sequence**, and **for every call** in a sequence, LTree and `LTreeGood` computes the “same” result

Lets Start Thinking About Writing Models

- A set of test sequences:

```
Seq1: T=new LTree(); root=T.addRoot(2); ...; T.replace(root,4); ..
Seq2: T=new LTree(); root=T.addRoot(8); ...; addChild(5,root); ...
...
Seq1000: ...
```

- Computing the “same” (Seq1 example):

```
T=new LTree();               T’=new LTreeGood();
// no relation between Ts

root=T.addRoot(2);           root’=T’.addRoot(2);
// no relation between roots

...

T.replace(root,4);           T’.replace(root’,4);
// both should return 2!
...
```

Writing and Checking a Model

- Ok, so how do we:

Writing and Checking a Model

- Ok, so how do we:

    1. Generate test sequences?

Writing and Checking a Model

- Ok, so how do we:

  1. Generate test sequences?
  2. Write a new model LTreeGood and

Writing and Checking a Model

- Ok, so how do we:

  1. Generate test sequences?
  2. Write a new model LTreeGood and
  3. **Systematically** check LTreeGood against LTree?

Writing and Checking a Model

- We could write the LTreeGood model in Java

Writing and Checking a Model

- We could write the LTreeGood model in Java
- **But** Java is not a very compact language (we have to write many lines of code), not very elegant

Writing and Checking a Model

- We could write the LTreeGood model in Java
- **But** Java is not a very compact language (we have to write many lines of code), not very elegant
- So we choose Erlang, concretely the QuickCheck for Erlang random testing tool

Writing and Checking a Model

- We could write the LTreeGood model in Java
- **But** Java is not a very compact language (we have to write many lines of code), not very elegant
- So we choose Erlang, concretely the QuickCheck for Erlang random testing tool
- Since no Erlang lecture yet, running example will have to wait...

Tree Model

- Our Tree model is **state based**
- What is the state?
  A **set** of pairs $\langle \mathit{TreeIdentifier}, \mathit{Tree} \rangle$
  (a $\mathit{TreeIdentifier}$ is a reference)
- What is a tree?
  Some nodes (references) connected in a tree-like data structure

Tree Model

- Our Tree model is **state based**
- What is the state?
  A **set** of pairs $\langle \mathit{TreeIdentifier}, \mathit{Tree} \rangle$
  (a $\mathit{TreeIdentifier}$ is a reference)
- What is a tree?
  Some nodes (references) connected in a tree-like data structure
- We show how each method changes the state:
  `new Tree()` applied in a state S causes a new state S'
  (where a new tree has been added)

Tree Model

For every method (such as `new Tree()`) we define three functions:

- A precondition (in what states is the method **applicable**) – useful for excluding non-interesting test cases
- A "postcondition" defining what the method call should return (or an exception), for comparing with the **LTree** implementation
- The "next state" defining what is the state after executing the method call

Tree Model Example (precondition)

- Let us consider the operation

  `t.replace(Position<Integer> n, Integer v)`

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)
- We assume that the current state is S
- **Precondition:**

Tree Model Example (precondition)

- Let us consider the operation

```
t.replace(Position<Integer> n, Integer v)
```

applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)

- We assume that the current state is S
- **Precondition:** $\mathtt{t} \in \mathit{trees}(S)$ and $\mathtt{n} \in \mathit{nodes}(\mathtt{t})$
- Why?

Tree Model Example (precondition)

- Let us consider the operation

```
t.replace(Position<Integer> n, Integer v)
```

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)

- We assume that the current state is S
- **Precondition:** $\texttt{t} \in \mathit{trees}(S)$ and $\texttt{n} \in \mathit{nodes}(\texttt{t})$
- Why? We do not want to test:

```
t1 = new Tree();
t2 = new Tree();
p1 = t1.addRoot(10);
p2 = t2.addRoot(20);
t1.replace(p2,15);
```

Tree Model Example (postcondition)

- Let us consider the operation

  `t.replace(Position<Integer> n, Integer v)`

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)
- We assume that the current state is S
- **Postcondition:**

Tree Model Example (postcondition)

- Let us consider the operation

  `t.replace(Position<Integer> n, Integer v)`

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)

- We assume that the current state is S
- **Postcondition:** `returnValue` $= \mathit{nodeValue}(n, S)$
- That is, we check that actual result value from applying `result(n,v)` to LTree is the same value as our state (S) has

Tree Model Example (postcondition)

- Let us consider the operation

  `t.replace(Position<Integer> n, Integer v)`

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)
- We assume that the current state is S
- **Postcondition:** `returnValue` $= \mathit{nodeValue}(n, S)$
- That is, we check that actual result value from applying `result(n,v)` to LTree is the same value as our state (S) has
- That is, we check that our model LTreeGood == LTree on the `result` operation in state S

Tree Model Example (next state)

- Let us consider the operation

  `t.replace(Position<Integer> n, Integer v)`

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)
- We assume that the current state is S
- **Next state:**

Tree Model Example (next state)

- Let us consider the operation

  ```
  t.replace(Position<Integer> n, Integer v)
  ```

  applied to the tree object `t` (which replaces the value of the node `n` with the integer `v` and returns the old value)
- We assume that the current state is S
- **Next state:** $S' = updateNodeValue(v, n, S)$
- That is, we replace the value associated with the node in our (LTreeGood) state S using the operation $updateNodeValue(v, n, S)$, resulting in a new state S'

Generating Test Data

- Finally we have to generate test data (a lot of test sequences)

```
Seq1: T=new LTree(); root=T.addRoot(2); ...; T.replace(root,4); ..
Seq2: T=new LTree(); root=T.addRoot(8); ...; addChild(5,root); ...
...
Seq1000: ...
```

- We do that using a "symbolic state" which remembers the operations we have applied, and the "symbolic" return values (using preconditions and next state definitions)

Generating Test Data

- Finally we have to generate test data (a lot of test sequences)

```
Seq1: T=new LTree(); root=T.addRoot(2); ...; T.replace(root,4); ..
Seq2: T=new LTree(); root=T.addRoot(8); ...; addChild(5,root); ...
...
Seq1000: ...
```

- We do that using a "symbolic state" which remembers the operations we have applied, and the "symbolic" return values (using preconditions and next state definitions)
- Example: the operation `new Tree()` can always be included in a test sequences

Generating Test Data

- Finally we have to generate test data (a lot of test sequences)

```
Seq1: T=new LTree(); root=T.addRoot(2); ...; T.replace(root,4); ..
Seq2: T=new LTree(); root=T.addRoot(8); ...; addChild(5,root); ...
...
Seq1000: ...
```

- We do that using a “symbolic state” which remembers the operations we have applied, and the “symbolic” return values (using preconditions and next state definitions)
- Example: the operation `new Tree()` can always be included in a test sequences
- Example: the operation `replace(v,n)` can be included in a test sequence only after there is a tree `t`, and a node `n`, but for any integer `n` in the tree
- ...

What have we obtained?

- A nice declarative description of the LTree class

What have we obtained?

- A nice declarative description of the LTree class
- We can generate an arbitrary number of test sequences **automatically** – typically leads to much more thorough testing than using test metrics

What have we obtained?

- A nice declarative description of the LTree class
- We can generate an arbitrary number of test sequences **automatically** – typically leads to much more thorough testing than using test metrics
- We can check each test sequence **automatically**

What have we obtained?

- A nice declarative description of the LTree class
- We can generate an arbitrary number of test sequences **automatically** – typically leads to much more thorough testing than using test metrics
- We can check each test sequence **automatically**
- Drawbacks: we have to write the model
- Normally 50% bugs found in model, 50% bugs found in tested implementation
  (but probably same for normal testing)