# Generating Representation Invariants of Structurally Complex Data

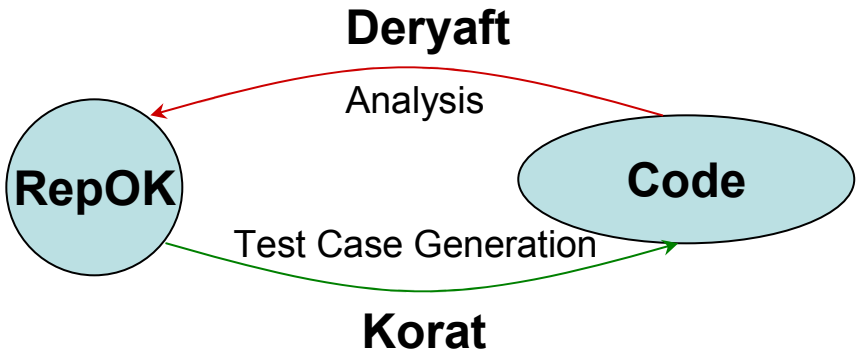
Muhammad Zubair Malik
University of Texas at Austin

# Motivation

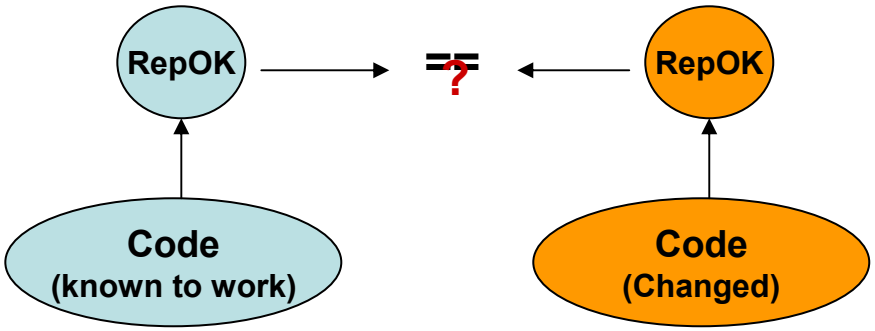
- Invariants are useful to:
  - Understand programs
  - Refine specifications to programs
  - Prove programs correct (Floyd-Hoare logics)
- However, it can be a burden convincing programmers to write them

# Why Discover Invariants?

- Fully Automate the testing process




- Better understand your own code (some decisions might have unanticipated ramifications)



# Dynamic Invariant Detection

- Detect Automatically from code!
  - But…Invariants are difficult to detect statically (undecidable in some cases)
- Daikon: Along the execution of the program, output the value of variables in some points of interest. Then, try to relate those values by post-processing the log.
- Deryaft: View program heap as an edge labeled graph then evaluate for global and local properties of graph.

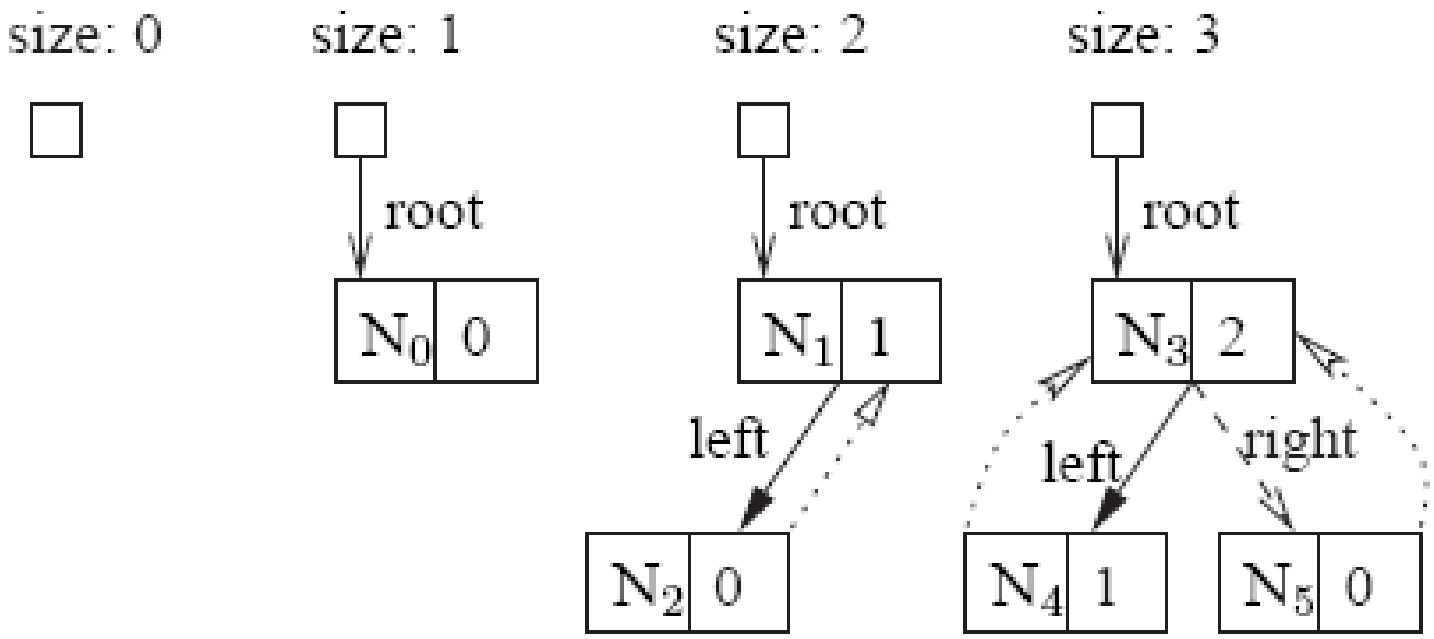
# Example: Binary Tree representation of Heap

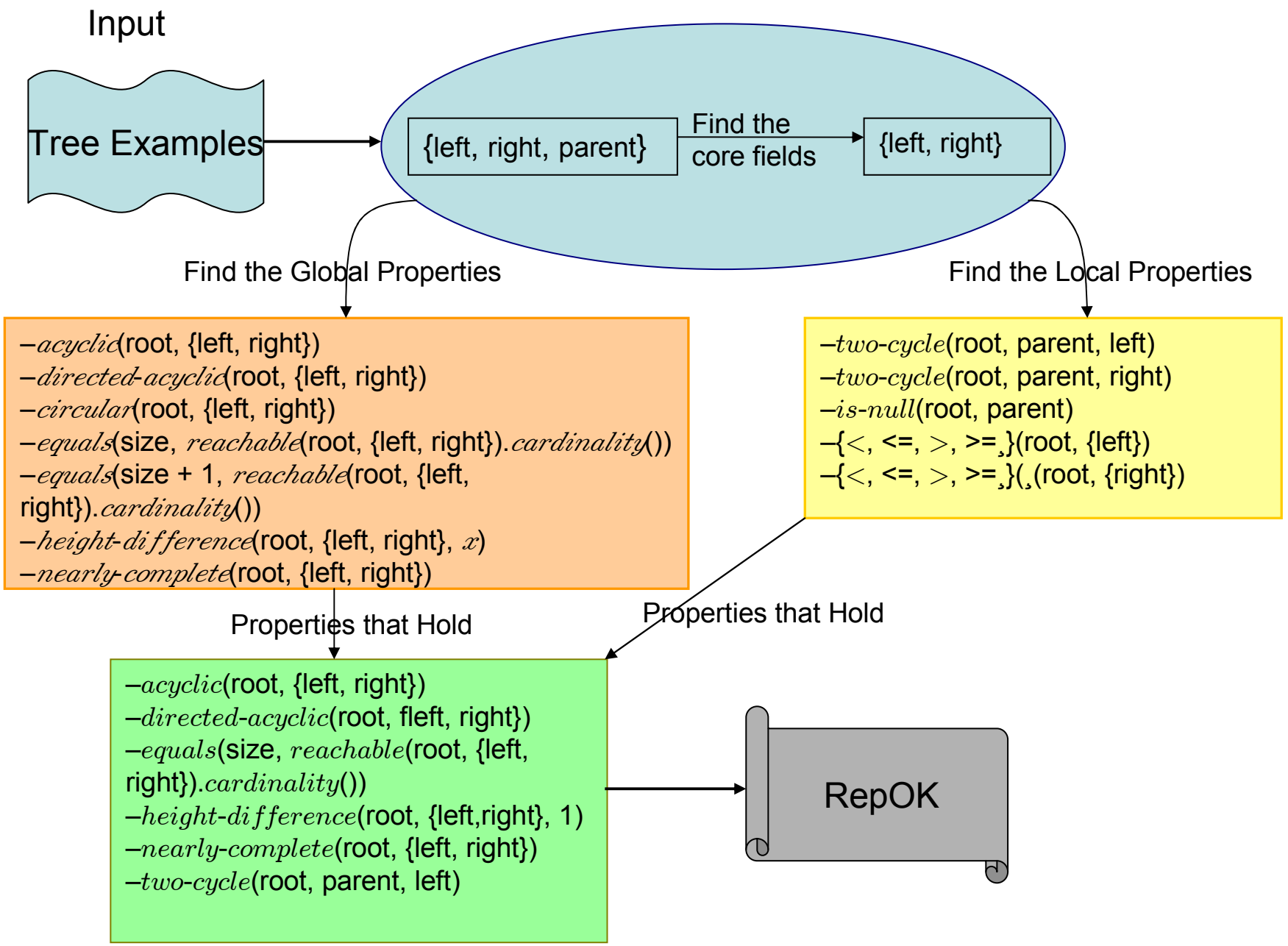
```
public class BinaryTree {
        Node root; // first node in the tree
        int size; // number of nodes in the tree

        private static class Node {
                Node left;
                Node right;
                Node parent;
                int key;
        }
}
```

**Invariants**

- Acyclicity along left and right
- Correctness of parent and size
- Heap property
- Nearly complete binary tree

# Example continued




Input
Tree Examples
{left, right, parent}
Find the core fields
{left, right}
Find the Global Properties
Find the Local Properties
–acyclic(root, {left, right})
–directed-acyclic(root, {left, right})
–circular(root, {left, right})
–equals(size, reachable(root, {left, right}).cardinality())
–equals(size + 1, reachable(root, {left, right}).cardinality())
–height-difference(root, {left, right}, x)
–nearly-complete(root, {left, right})
–two-cycle(root, parent, left)
–two-cycle(root, parent, right)
–is-null(root, parent)
–{<, <=, >, >=¸}(root, {left})
–{<, <=, >, >=¸}(¸(root, {right})
Properties that Hold
Properties that Hold
–acyclic(root, {left, right})
–directed-acyclic(root, fleft, right})
–equals(size, reachable(root, {left, right}).cardinality())
–height-difference(root, {left,right}, 1)
–nearly-complete(root, {left, right})
–two-cycle(root, parent, left)
RepOK

## Output for Tree Example

```
public boolean repOk() {
  if (!acyclicCore(root)) return false;
  if (!sizeOk(size, root)) return false;
  if (!nearlyComplete(root)) return false;
  if (!parentNull(root)) return false;
  if (!parentTwoCycleLeft(root)) return false;
  if (!parentTwoCycleRight(root)) return false;
  if (!greaterThanLeft(root)) return false;
  if (!greaterThanRight(root)) return false;
  return true;
}
```

## Predicate Function

```
private boolean parentNull(Node n) {
  return (n.parent == null);
}
```

## A More Interesting Predicate Function

```
private boolean parentTwoCycleLeft(Node n) {
  Set<Node> visited = new HashSet<Node>();
  LinkedList<Node> worklist = new LinkedList<Node>();
  if (n != null) {
    worklist.addFirst(n);
    visited.add(n);
  }
  while (!worklist.isEmpty()) {
    Node current = worklist.removeFirst();
    if (current.left != null) {
      if (current.left.parent != current) return false;
      if (visited.add(current.left)) {
        worklist.addFirst(current.left);
      }
    }
    if (current.right != null) {
      if (visited.add(current.right)) {
        worklist.addFirst(current.right);
      }
    }
  }
  return true;
}
```

# Deryaft Algorithm

- Identification of core and derived fields
- Formulation of global and local properties that are relevant
- Computation of properties that actually hold
- Minimization of properties
- Generation of Java code that represents properties.

# Program heap as an Edge-Labeled Graph

- The heap of a Java program as an edge-labeled directed graph
  - nodes represent objects
  - edges represent fields
- The presence of an edge labeled *f* from node *o* to *v* says that the *f* field of the object *o* points to the object *v* (or is null) or has the primitive value *v*.
- Graph as a set of nodes and a collection of relations

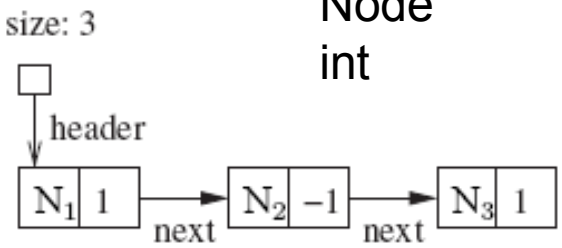
# Graph View of the Programs

- partition the set of nodes according to the declared classes
- partition the set of edges according to the declared fields
- represent null as a special node
- program state is represented by an assignment of values to these sets and relations
- Deryaft focuses on generating properties of such graphs, including properties that involve reachability, e.g., acyclicity

```
public class SinglyLinkedList {
  private Node header; // first list node
  private int size; // number of nodes in the list
  private static class Node {
    int elem;
    Node next;
  }
}
```

primitive type
SinglyLinkedList
Node
int

four relations
header: SinglyLinkedList x Node
size: SinglyLinkedList x int
elem: Node x int
next: Node x Node




SinglyLinkedList = { L0 }
Node = { N1, N2, N3 }
int = { -1, 0, 1 }

header = { <L0, N0> }
size = { <L0, 3> }
elem = { <N1, 1>, <N2, -1>, <N3, 0> }
next = { <N1, N2>, <N2, N3>, <N3, null> }

# Core and derived fields

- *A subset $C \subseteq F$ is a core set with respect to S if for all structures $s \in S$, the set of nodes reachable from the root r of s along the fields in C is the same as the set of nodes reachable from r along the fields in F.*
  - {left, right, parent} → {left, right}
- *For a core set C, the set of fields F-C is a derived set.*
  - {left, right, parent}- {left, right} → {parent}

# Properties of interest

- properties of rooted edge-labeled directed graphs which are likely representatives of structurally complex data
  - *Global*
  - *Local*
- Checking properties at other scales is computationally infeasible
- **Global: Reachability**
  - properties include the *shape* of the structure reachable from root along some set of reference fields
  - The shapes can be
    - *Acyclic*
    - *Directed-Acyclic*
    - *Circular*
    - *Arbitrary*.

- **Global: primitive fields**
  - properties relating values of integer fields and cardinalities of sets of reachable objects
- **Global: path lengths**
  - relate lengths of different paths from root (over non-linear structures, such as trees)
  - Relate lengths of different paths from root
    - *Balanced*
    - *height-balanced*

- **Local: reference fields**
  - local properties that relate different types of edges
    - *two-cycles*
- **Local: primitive fields**
  - primitive values that relate a node's value to it's successors along reference fields
    - in a binary tree, the value in a node might be greater than the values in the node's children

# Experiments

- Diverse examples
  - structures library (Collection Classes)
  - standalone application
- Construct representative structures (we used five per structure class)
- Deryaft correctly generated all the standard data structure invariants for following structures:

- **Singly-linked acyclic list**
  - A list object has a header node; each list node has a next field
  - Integrity constraint: is acyclicity along next
- **Ordered list**
  - singly-linked acyclic list, whose nodes have integer elements
  - Integrity constraints are:
    - acyclicity
    - ordering on the elements (ascending or descending)
- **Doubly-linked circular list (**java.util.LinkedList)
  - A list object has a header node
  - each list node has a next and a previous field
  - Integrity constraints
    - circularity along next
    - transpose relation between next and previous

- **Binary search tree**
  - A binary search tree object has a root node
  - each node has a left and a right child node, a parent, and an integer key
  - Integrity constraints are
    - acyclicity along left and right
    - correctness of parent
    - correct ordering of keys
    - for each node, its key is larger than any of the keys in the left sub-tree and smaller than any of the keys in the right-sub tree
- **AVL tree**
  - a height-balanced binary search tree
  - Integrity constraints are
    - All the binary search tree constraints
    - the height-balance constraint

- **Heap array (priority queue)**
  - an array-based implementation of the binary heap data structure
  - heap has a capacity that is the length of the underlying array
  - a size that is the number of elements currently in the heap
  - Integrity constraints are:
    - size <= capacity
    - Heap property: an element is larger than both its children
- **Intentional name**
  - INS is a service location system
  - specify *what* you are looking for without having to know *where* it may be situated in a dynamic network
  - Data Structure: a hierarchical arrangement of *attribute-value pairs*
  - implemented using the class AVPair that has two String fields *attribute* and *value* and a Vector<AVPair> field children
  - Integrity constraints are:
    - attribute and value of the root are null
    - the children of a node have unique attributes
    - the structure is acyclic along the children field

# Limitations

- We are only generating likely program invariants.
- The underlying problem is undecidable
- The analysis cannot be sound and complete
- The example structures may not be a good representative of the class structures
- conjecturing all possible relations among integer fields is infeasible

# More issues

- Optimization: The repOk code that Deryaft outputs typically performs several traversals over a given structure, minimize the traversals
- Extensibility:
  - Be able to introduce new invariant checks in existing system
  - Annotation language to express more richer class of invariants
- Integration with other existing invariant generators (Daikon)
- Integrate static analysis for faster generation of invariants

# Future Work

- Benefit from the underlying graph view of Deryaft
- Use the spectral representation of graphs (Shape from Spectra) to optimize detection
- Use the graph-cuts to separate various connected structures (A hash table with pointers to a doubly linked list)

# Conclusion

- Dynamically detecting likely invariants is feasible
- Given a small set of concrete structures we can generate *representation invariants of structurally complex data*
- Detecting core fields allows us to focus on invariants of interest
- Our approach heuristically restricts search over global and local properties (checking all scales is infeasible)

Questions!