**Problem:** Static Deadlock Prevention

Deadlock is a cycle of threads waiting for resources held by the next thread in the cycle.

Techniques exist that can check dangerous code like the following:

```
Object a in A;  LockLevel A;
Object b in B;  LockLevel B > A;
lock (a) {   lock (b) {
  lock (a) {
  ...
  }  }
}  }
```

But they cannot check code like this, because they require a total ordering on all locks acquired:

```
BinaryTreeNode node;
...
lock (node) {
  lock (node.left) {
    lock (node.right) {
      ...
    }  }
}  And most cannot check code that reorders locks
```

**Solution:** A Different View of Lock Ordering

- Locking a node can grant permission to lock the children
- **Insight:** This permission-granting relation is often implicitly defined by heap references
- **Insight:** Mutating the structure changes effective locking order
- **Insight:** If the only thread that can lock the children while holding other locks holds the parent, both children can be locked in *either* order

Static Lock Capabilities

- Statically prevent deadlock
- Partitioning by capability
- Taking one lock grants static capability to acquire other locks
  - Each object may be locked first OR while one specific other lock is held
- Order among capabilities defined by heap edges
- Allow locking arbitrary contiguous subtrees of recursive structures

**A Type System for Lock Capabilities**

**Challenges**

- Aliasing
- Isolating ordering
- Reordering
- Changing capability association
- Cycles

**Solutions**

- Uniqueness, borrowing, and partial references
- **Unique refs associate an object with 1 cap**
- **Partial refs carry no capability information**
- **Naturalize capability to <= 1 thread**
- **Easily update capability assoc.**
- **Allow locking arbitrary contiguous locks acquired (see right)**

**Comparison to Prior Work (Lock Levels)**

- **Lock Levels**
  - Requires total ordering among all locks acquired
  - All threads use same ordering
  - At best 1 narrow path in trees
  - Fixed levels & ordering
  - Easily checks locking order for distant pointers
  - Well-suited to layering subsystems
  - 2 systems can verify splay trees:
    - Chalice (full program verification)
    - SafeJava (awkward, unspecified extension to lock levels)

- **Lock Capabilities**
  - Requires <= 1 object grants cap. on each object, grant relation is acyclic
  - Notion of “safe to acquire” restricted to 1 thread
  - Arbitrary contiguous subtrees
  - Capability per object, reordered by heap references
  - See caps for only locally-related pointers
  - Well-suited for fine-grained synchronization
  - Can verify splay trees
  - Can verify structures w/o total ordering on locks acquired (see right)

**Going Further: Circular Lists**

- **Benchmarks:**
  - Circular lists

- **Premil:**
  - Chalice (full program verification)
  - SafeJava (awkward, unspecified extension to lock levels)

- **Solution:**
  - **Lock any 1 node OR**
    - First lock “whole list” object, then any nodes in any order
  - **Not verifiable with any prior work**

**Splay Trees**

```
class Node {
    Node left, right;
    synchronized void rotateRight() {
        Node x = this.right;
        if (x == null) return;
        synchronized(x) {
            if (x.left == null) return;
            synchronized(x.left) {
                unique Node x = dread(x.left);
                unique Node w = x.right;
                v.right = w;
                x.right = w;
                w = null;
            }
        }
    }
    synchronized void rotateLeft() {
        Node x = this.left;
        if (x == null) return;
        synchronized(x) {
            if (x.right == null) return;
            synchronized(x.right) {
                unique Node x = dread(x.right);
                unique Node w = x.left;
                v.left = w;
                x.left = w;
                w = null;
            }
        }
    }
}
```