Static Lock Capabilities for Deadlock-Freedom

Colin S. Gordon
csgordon@cs.washington.edu

University of Washington

TLDI, January 28, 2012
Joint work with Michael D. Ernst and Dan Grossman

Verifying Deadlock Freedom

Deadlock

A cycle of threads, each blocked waiting for a resource held by the next thread in the cycle.

$$T_1 \to T_2 \to \ldots \to T_n, \qquad T_1 = T_n$$

Goal

Statically verify deadlock freedom for fine-grained locking

- Balanced binary trees
- Resizable hash tables
- Array elements
- Circular lists

Approach

A static (capability) type system

Deadlock-Free Code

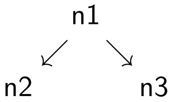
Assuming n2 == n1.left and n3 == n1.right:

Thread1 : sync n2 {}

Thread2 : sync n3 {}

Thread3 : sync n1 {sync n1.left {sync n1.right {}}}

Thread4 : sync n1 {sync n1.right {sync n1.left {}}}



Deadlock-Free Code

Assuming n2 == n1.left and n3 == n1.right:

Thread1 : sync n2 {}
Thread2 : sync n3 {}
Thread3 : sync n1 {sync n1.left {sync n1.right {}}}
Thread4 : sync n1 {sync n1.right {sync n1.left {}}}

n1
$\swarrow$ $\searrow$
n2 n3

Prior static approaches require either:

- A total ordering on `n1`'s children (rejects T3 or T4), or
- Disallow interior pointers (`n2`, `n3`, rejecting T1 and T2)

> Lock capabilities impose neither restriction.

Lock Capabilities

Lock Capability

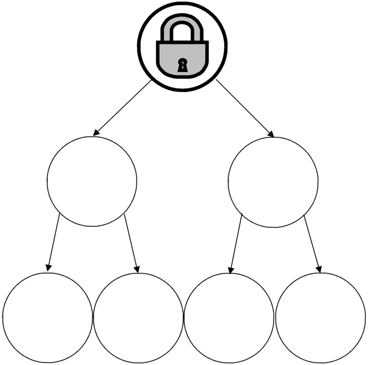
A static capability that permits acquiring additional locks

- Baked into a type-and-effect system
- Proved sound (they prevent deadlock)
- Straightforward extensions
- Scale to handle a set of diverse structures
  - with the help of some extensions to plumb singleton types

Intuition: Tree-Based Ordering

Fine-grained locking in a binary tree:

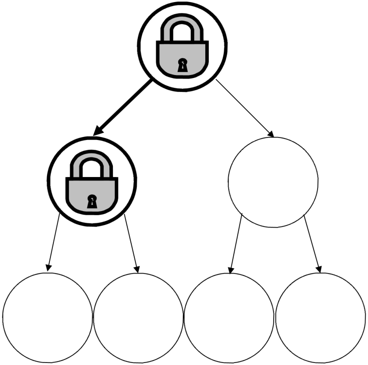
- Acquiring one lock while holding none avoids deadlock;
  "First lock is free"



Intuition: Tree-Based Ordering

Fine-grained locking in a binary tree:

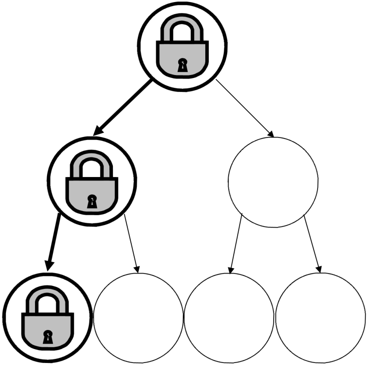
- Acquiring one lock while holding none avoids deadlock;
  "First lock is free"
- Following *tree order* deeply through the tree avoids deadlock.



Intuition: Tree-Based Ordering

Fine-grained locking in a binary tree:

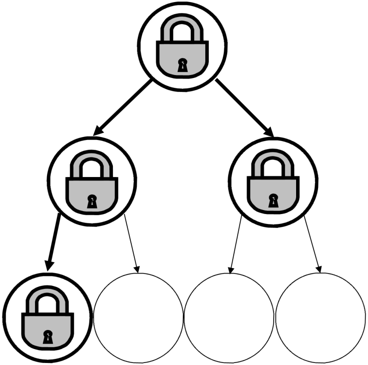
- Acquiring one lock while holding none avoids deadlock; "First lock is free"
- Following *tree order* deeply through the tree avoids deadlock.



Intuition: Tree-Based Ordering

Fine-grained locking in a binary tree:

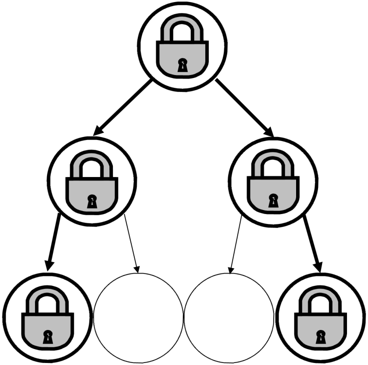
- Acquiring one lock while holding none avoids deadlock; "First lock is free"
- Following *tree order* deeply through the tree avoids deadlock.
- Assuming children are acquired only while holding the parent lock, *locking siblings avoids deadlock*.



Intuition: Tree-Based Ordering

Fine-grained locking in a binary tree:

- Acquiring one lock while holding none avoids deadlock; "First lock is free"
- Following *tree order* deeply through the tree avoids deadlock.
- Assuming children are acquired only while holding the parent lock, *locking siblings avoids deadlock*.



Generalizing Beyond Trees

Trees → Tree-shaped Partial Orders

In an *immutable* tree-shaped partial ordering, a thread may acquire a lock l when:

- It holds no other locks, or
- It holds a lock l' and l is a child of l'

Generalizing Beyond Trees

Trees → Tree-shaped Partial Orders

In an *immutable* tree-shaped partial ordering, a thread may acquire a lock l when:

- It holds no other locks, or
- It holds a lock l' and l is a child of l'

Notice:

- No ordering imposed between siblings
- No restriction on aliases

Generalizing Beyond Trees

Trees → Tree-shaped Partial Orders

In an *immutable* tree-shaped partial ordering, a thread may acquire a lock l when:

- It holds no other locks, or
- It holds a lock l' and l is a child of l'

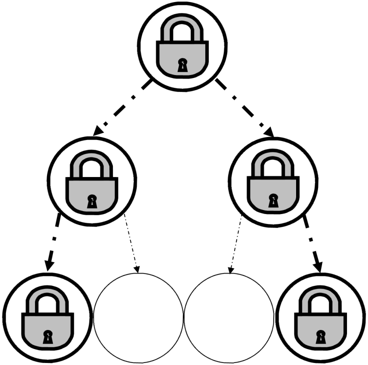
Notice:

- No ordering imposed between siblings
- No restriction on aliases

Harder:

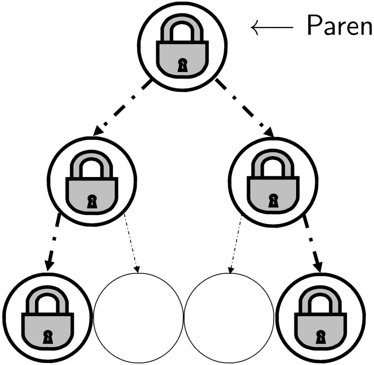
- Early lock releases
- Modifying the partial order

Lock Capabilities



```
class TreeNode {
  guardedBy⟨this⟩ TreeNode left;
  guardedBy⟨this⟩ TreeNode right;
}
```

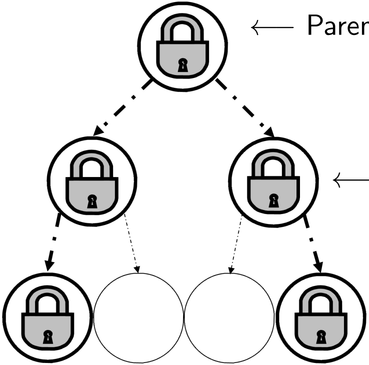
Lock Capabilities



$\longleftarrow$ Parent lock x *grants* capability $\langle x \rangle$

```
class TreeNode {
  guardedBy⟨this⟩ TreeNode left;
  guardedBy⟨this⟩ TreeNode right;
}
```

Lock Capabilities

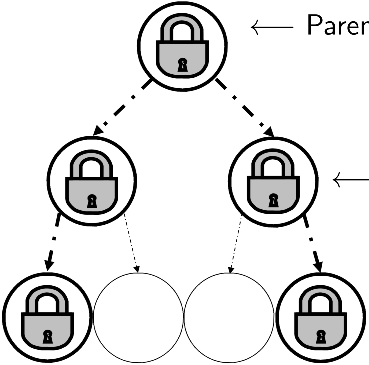


⟵ Parent lock x *grants* capability $\langle x \rangle$

⟵ Child type includes the *guarding capability*:
$x.right$: guardedBy$\langle x \rangle$ TreeNode

```
class TreeNode {
  guardedBy⟨this⟩ TreeNode left;
  guardedBy⟨this⟩ TreeNode right;
}
```

Lock Capabilities



$\longleftarrow$ Parent lock x *grants* capability $\langle x \rangle$

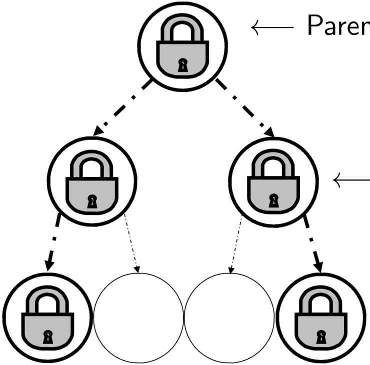
$\longleftarrow$ Child type includes the *guarding capability*: $x.right$: guardedBy$\langle x \rangle$ TreeNode

lock (x) in lock ($x.right$) in ...

May only acquire lock of type guardedBy$\langle x \rangle$ when holding lock x (or no locks at all).

```
class TreeNode {
  guardedBy⟨this⟩ TreeNode left;
  guardedBy⟨this⟩ TreeNode right;
}
```

Lock Capabilities



← Parent lock x *grants* capability $\langle x \rangle$

← Child type includes the *guarding capability*:
$x.right$: guardedBy$\langle x \rangle$ TreeNode

lock (x) in lock $(x.right)$ in ...

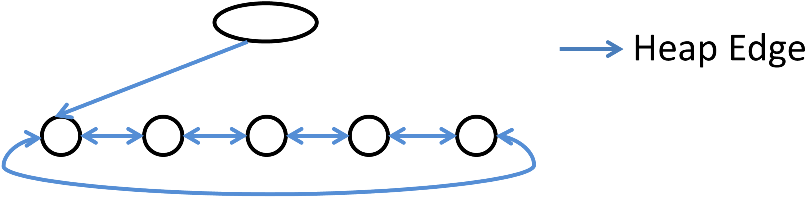
May only acquire lock of type guardedBy$\langle x \rangle$ when holding lock x (or no locks at all).

```
class TreeNode {
  guardedBy⟨this⟩ TreeNode left;
  guardedBy⟨this⟩ TreeNode right;
}
```

Deadlock freedom follows from the *capability granting relation* being a forest

Structures with Cycles

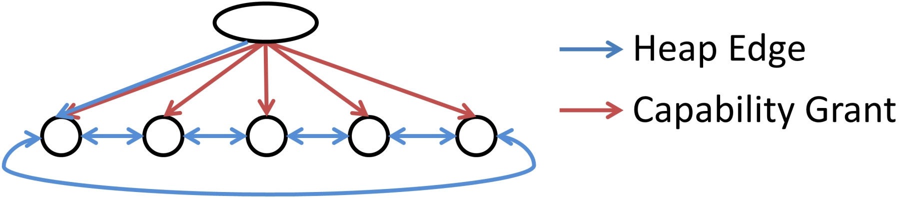
A forest-shaped capability granting relation doesn't require forest-shaped data structures. For example, here is a circular list:



This circular list has cycles in the heap, but a tree-shaped capability granting relation.

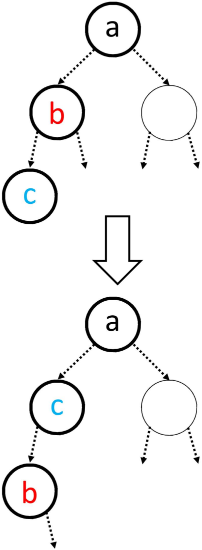
Structures with Cycles

A forest-shaped capability granting relation doesn't require forest-shaped data structures. For example, here is a circular list:



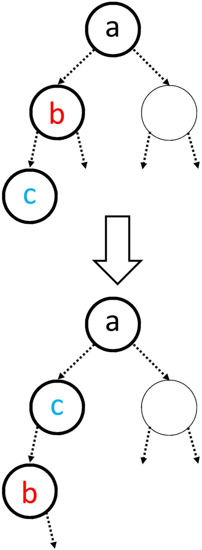
This circular list has cycles in the heap, but a tree-shaped capability granting relation.

Supporting Mutable Structures



Lock relationships can change dynamically, so we need:

Supporting Mutable Structures

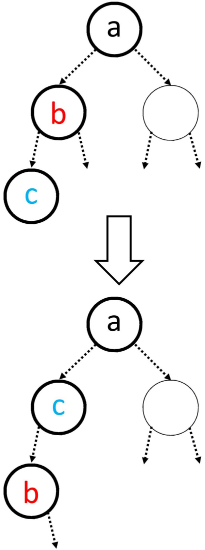


Lock relationships can change dynamically, so we need:

- Strong Updates

  $\Longrightarrow$ weakened form of uniqueness

Supporting Mutable Structures



Lock relationships can change dynamically, so we need:

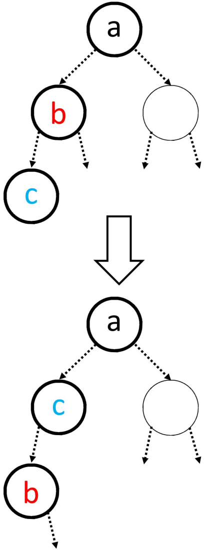
- Strong Updates

  $\Longrightarrow$ weakened form of uniqueness

- Preserving Acyclicity

  $\Longrightarrow$ track shape of capability-granting relation

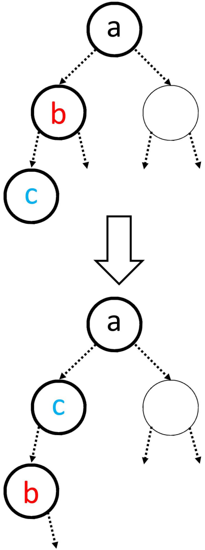
Supporting Mutable Structures



Lock relationships can change dynamically, so we need:

- Strong Updates
  $\Longrightarrow$ weakened form of uniqueness
- Preserving Acyclicity
  $\Longrightarrow$ track shape of capability-granting relation
- Releasing Out-Of-Order
  $\Longrightarrow$ restrictions on lock acquisition

Supporting Mutable Structures



Lock relationships can change dynamically, so we need:

- Strong Updates
  - $\Longrightarrow$ weakened form of uniqueness
- Preserving Acyclicity
  - $\Longrightarrow$ track shape of capability-granting relation
- Releasing Out-Of-Order
  - $\Longrightarrow$ restrictions on lock acquisition
  - ▸ No time to discuss out-of-order releases

Changing Capability Grants

The *capability granting relation* that determines each lock's guard must allow changes.

Partial Uniqueness

A single reference carries the guard information for an object

1	u_guardedBy$\langle x \rangle$ TreeNode	"Unique" with guard information
∞	guardless TreeNode	Duplicable, no guard information

Changing Capability Grants

The *capability granting relation* that determines each lock's guard must allow changes.

Partial Uniqueness

A single reference carries the guard information for an object

1	u_guardedBy⟨x⟩ TreeNode	"Unique" with guard information
∞	guardless TreeNode	Duplicable, no guard information

Partial Strong Updates

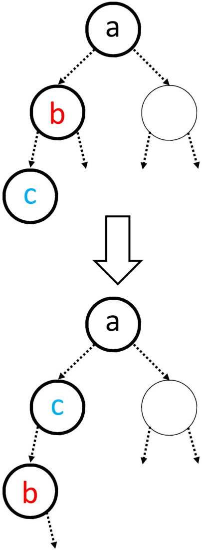
Guard information is isolated, enabling strong updates to the guard

$$x : \text{u\_guardedBy}\langle y\rangle\ \text{TreeNode} \longrightarrow x : \text{u\_guardedBy}\langle z\rangle\ \text{TreeNode}$$

Goal: Type system infers strong updates without explicit guidance

Changing Tree Structure

```
public TreeNode {
    public u_guardedBy⟨this⟩ TreeNode left;
    public u_guardedBy⟨this⟩ TreeNode right;
}
...
guardless TreeNode a;
...
lock(a) {
  lock(a.left) {
    lock(a.left.left) {
      let b = dread(a.left) in
      let c = dread(b.left) in
      c.left := dread(b);
      a.left := dread(c);
} } }
```



Destructive Reads

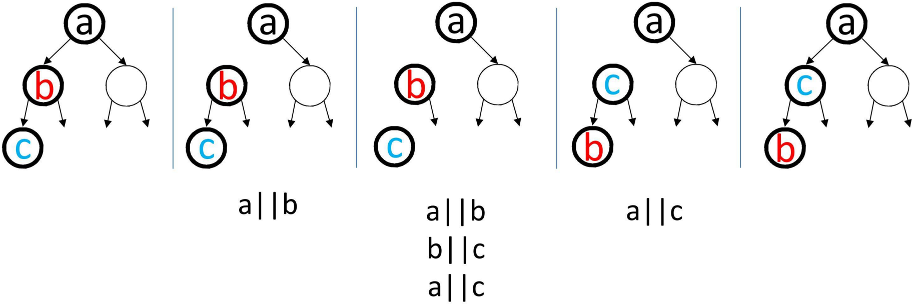
dread(p) atomically assigns null to path p and returns the old value, preventing duplication.

Preserving Acyclicity

Changes to the capability-granting relation must not create cycles.

We track disjointness of capability-granting trees in a flow-sensitive manner.

- Removing an edge produces two mutually disjoint trees
- Adding an edge between two mutually disjoint trees produces one tree



The Core Type System

Core typing judgement:

$$\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$$

The Core Type System

Core typing judgement:

tree disjointness

$$\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$$

The Core Type System

Core typing judgement:

tree disjointness

$$\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$$

local variable typing

The Core Type System

Core typing judgement:

tree disjointness

$$\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$$

local variable typing

held locks ($\equiv$ capabilities)

The Core Type System

Core typing judgement:

tree disjointness

$$\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$$

local variable typing

held locks ($\equiv$ capabilities)

Two theorems proven for basic lock capabilities with reordering:

1. Type Preservation
   - Long, straightforward
2. Deadlock Freedom Preservation
   - Extended semantics with capability-use log in graph form, modeling thread dependencies

Proposed Extensions

"Plumbing" Extensions:

- Arrays (treated as object with integer-named fields)

Proposed Extensions

"Plumbing" Extensions:

- Arrays (treated as object with integer-named fields)
- Fixed guards (no strong update, but sharing guard info)

Proposed Extensions

"Plumbing" Extensions:

- Arrays (treated as object with integer-named fields)
- Fixed guards (no strong update, but sharing guard info)
- External capabilities
  - Parameterized classes a la RCC/Java
  -
```
class CircularListNode<ghost List l> {
  fixed_guard<l> CircularListNode<l> next;
  fixed_guard<l> CircularListNode<l> prev;
  ...
}
```

Proposed Extensions

"Plumbing" Extensions:

- Arrays (treated as object with integer-named fields)
- Fixed guards (no strong update, but sharing guard info)
- External capabilities
  - Parameterized classes a la RCC/Java
  -
    ```
    class CircularListNode<ghost List l> {
      fixed_guard<l> CircularListNode<l> next;
      fixed_guard<l> CircularListNode<l> prev;
      ...
    }
    ```

More substantial extensions:

- Unstructured Locking (requires more precise capability tracking)

Proposed Extensions

"Plumbing" Extensions:

- Arrays (treated as object with integer-named fields)
- Fixed guards (no strong update, but sharing guard info)
- External capabilities
  - Parameterized classes a la RCC/Java
  -
```
class CircularListNode<ghost List l> {
  fixed_guard<l> CircularListNode<l> next;
  fixed_guard<l> CircularListNode<l> prev;
  ...
}
```

More substantial extensions:

- Unstructured Locking (requires more precise capability tracking)
- Combination with *lock levels*

Examples

- Splay Tree Rotation
  - Captured by SafeJava and Chalice, but SafeJava special-cases

Examples

- Splay Tree Rotation
  - Captured by SafeJava and Chalice, but SafeJava special-cases
- Array Element Locking (with array extension)
  - Only addressed by Gadara, which may over-synchronize

Examples

- Splay Tree Rotation
  - Captured by SafeJava and Chalice, but SafeJava special-cases
- Array Element Locking (with array extension)
  - Only addressed by Gadara, which may over-synchronize
- Circular Lists (with external capabilities and fixed guard extensions)
  - List used in OS kernels
  - Each list node guarded by a central list object
  - Allows parallelism between threads using single nodes and one thread using multiple

Examples

- Splay Tree Rotation
  - Captured by SafeJava and Chalice, but SafeJava special-cases
- Array Element Locking (with array extension)
  - Only addressed by Gadara, which may over-synchronize
- Circular Lists (with external capabilities and fixed guard extensions)
  - List used in OS kernels
  - Each list node guarded by a central list object
  - Allows parallelism between threads using single nodes and one thread using multiple
- Dining Philosophers (with external capabilities, fixed guards, and explicit unlock)
  - All "chopstick" locks guarded by central lock
  - Threads "eat" by locking central lock, then chopsticks, then releasing central lock
  - Can build hierarchy of intermediate locks for improved parallelism

Examples

- Splay Tree Rotation
  - Captured by SafeJava and Chalice, but SafeJava special-cases
- Array Element Locking (with array extension)
  - Only addressed by Gadara, which may over-synchronize
- Circular Lists (with external capabilities and fixed guard extensions)
  - List used in OS kernels
  - Each list node guarded by a central list object
  - Allows parallelism between threads using single nodes and one thread using multiple
- Dining Philosophers (with external capabilities, fixed guards, and explicit unlock)
  - All "chopstick" locks guarded by central lock
  - Threads "eat" by locking central lock, then chopsticks, then releasing central lock
  - Can build hierarchy of intermediate locks for improved parallelism

All handled cleanly by a single general approach.

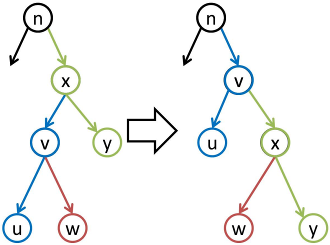
Contributions

- Introduced *lock capabilities*
  - New approach to verifying deadlock freedom
  - Well-suited to fine-grained locking
  - Suitable for any verification approach, we used types
- Proved soundness: lock capabilities ensure deadlock freedom
- Sketched useful, straightforward extensions
- Showed how lock capabilities can verify deadlock freedom for important, challenging examples

Backup Slides

Splay Tree Rotation

```
class Node {
    u_guardedBy<this>Node left;
    u_guardedBy<this>Node right;
}
...
  let final n = ... in
  lock (n) {
    let final x = n.right in
    if (x) {
      lock (x) {
        if (x.left) {
          let final v_name = x.left in
          lock (x.left) {
            let v = dread(x.left) in
            let final w_name = v.right in
            let w = dread(v.right) in
            // v.right := x
            v.right := dread(n.right);
            x.left := dread(w);
            n.right := dread(v);
  }}}}}
```



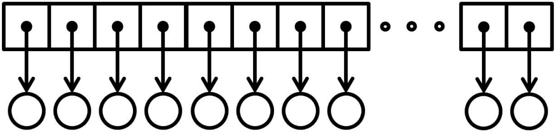
Differences from regular code are highlighted. Most can be inferred by a compiler.

Array-order Locking

Array-order locking is generally undecidable; lock capabilities enable a restricted form to be verified. In our core language extended with arrays and integers:

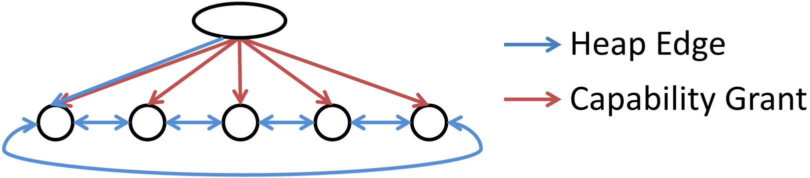
```
let final arr, unique a = new u_guardedBy Object[n] in
...
lock(arr) {
  lock(arr[i]) {
    lock(arr[j]) {
      ...
    }
  }
}
```

Note that we don't need to compare *i* and *j*!



Circular Lists

- The list of running processes in an OS kernel is *circular*
- It requires fine-grained locking for performance.
- Atomic resource transfer requires locking *multiple* processes.
- There is *no sensible ordering* on processes.



Orphaned Locks

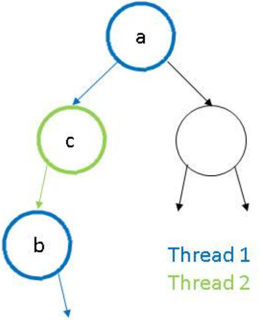
Acyclic capability granting is only half of soundness:

```
public TreeNode {
    public u_guardedBy⟨this⟩TreeNode left;
    public u_guardedBy⟨this⟩TreeNode right;
}
...
guardless TreeNode a;
...
lock(a) {
  lock(a.left) {
    lock(a.left.left) {
      let b = dread(a.left) in
      let c = dread(b.left) in
      c.left := dread(b);
      a.left := dread(c);
    } // release c
    lock(a.left) { // lock c again
      // do stuff
    }
  }
}
```

Orphaned Locks

Acyclic capability granting is only half of soundness:

```
public TreeNode {
    public u_guardedBy⟨this⟩TreeNode left;
    public u_guardedBy⟨this⟩TreeNode right;
}
...
guardless TreeNode a;
...
lock(a) {
  lock(a.left) {
    lock(a.left.left) {
      let b = dread(a.left) in
      let c = dread(b.left) in
      c.left := dread(b);
      a.left := dread(c);
    } // release c
    lock(a.left) { // lock c again
      // DEADLOCK!!!
    }
  }
}
```




```
lock(n) {            // lock c
  lock(n.left) {     // lock b
    // DEADLOCK!!!
  }
}
```

Theorem: Type Preservation

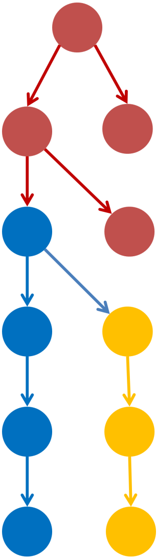
- Syntactic proof
- Extended typing rules add:
  - Heap typing Σ
  - Per-thread capability grants $\phi_i : \mathrm{Value} \rightarrow \mathrm{Variable}$ (or intuitively, $\mathrm{Lock} \rightarrow \mathrm{Lock}$)
- Requires many invariants
  - Most are natural (e.g. well-formed environments)
  - A few natural to preserve, subtle to state
    - e.g. relating multiple threads' assertions about the capability-granting relation
  - Full details in TR

Theorem: Deadlock Freedom Preservation

Deadlock freedom is a preservation proof:

- Build a labeled graph of how threads use capabilities
- Prove there is never a path between a single thread's locks using capabilities of multiple threads.

Detailed sketch in paper, full proof in TR.



Dining Philosophers

The problem:

- *The* canonical deadlock example
- n philosophers eating at a circular table
  - Only n chopsticks, one to each side of each philosopher
  - Must share chopsticks (locks) with neighbors
  - Philosophers are greedy and won't put down chopstick (release lock) until they've eaten
- There is no way to put a consistent structural ordering on chopsticks (locks)

Dining Philosophers

The problem:

- *The* canonical deadlock example
- *n* philosophers eating at a circular table
  - Only *n* chopsticks, one to each side of each philosopher
  - Must share chopsticks (locks) with neighbors
  - Philosophers are greedy and won't put down chopstick (release lock) until they've eaten
- There is no way to put a consistent structural ordering on chopsticks (locks)

With support for lock capabilities with unstructured locking:

- Capability-granting relation identical to the circular list
- With releasing "global" lock early:
  - Serializes acquisition
  - Allows parallelism between threads holding multiple locks
- *Verifiably* deadlock-free solution that allows some parallelism with simple code

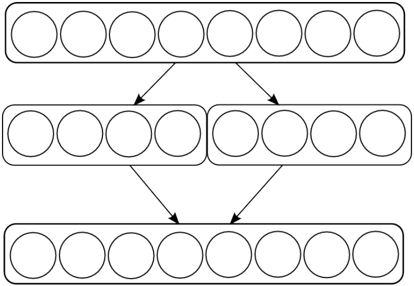
Background: Lock Levels

The program's locks are partitioned into *levels*, and the programmer specifies a partial order on levels.

Lock Levels Locking Protocol

A thread may acquire a lock *l* when:

- It holds no other locks, or
- *l* is in a lock level ordered *after* the level of *all* locks held



Limitations:

- Requires total ordering on any set of locks held concurrently.
- Can't deal with reordering, except for SafeJava and Chalice.

Comparing Lock Levels and Lock Capabilities

Fundamental philosophical difference: with lock levels, acquiring a lock *restricts* the set of locks the thread may then acquire, while with lock capabilities, acquiring a lock *extends* the set of locks the thread may then acquire.

Comparing Lock Levels and Lock Capabilities

Fundamental philosophical difference: with lock levels, acquiring a lock *restricts* the set of locks the thread may then acquire, while with lock capabilities, acquiring a lock *extends* the set of locks the thread may then acquire.

Lock Capabilities

- Are well-suited to fine-grained locking and reordering locks
- Allow some locking without total orderings
- Poorly-suited for locking unrelated "distant" locks

Lock Levels

- Are well-suited to locking unrelated "distant" locks
- Require total ordering on locks held simultaneously
- Poorly suited for fine-grained locking, or reordering locks
  - Except Chalice, which has a very smart variation

Comparing Lock Levels and Lock Capabilities

Fundamental philosophical difference: with lock levels, acquiring a lock *restricts* the set of locks the thread may then acquire, while with lock capabilities, acquiring a lock *extends* the set of locks the thread may then acquire.

Lock Capabilities

- Are well-suited to fine-grained locking and reordering locks
- Allow some locking without total orderings
- Poorly-suited for locking unrelated "distant" locks

Lock Levels

- Are well-suited to locking unrelated "distant" locks
- Require total ordering on locks held simultaneously
- Poorly suited for fine-grained locking, or reordering locks
  - Except Chalice, which has a very smart variation

It is possible to integrate the two for a more expressive system.

Chalice (Leino & Müller, ESOP'09, '10)

Combines a clever variant of levels with fractional permissions:

- Uses a *dense lattice* of levels, not discrete
  - For any levels l_0, l_1, exists l' s.t. $l_0 \sqsubset l' \sqsubset l_1$
- Uses fractional permissions on a ghost field μ to reorder

These add great flexibility over other lock level systems.

```
class TreeNode {
  TreeNode left, right;
  // declare full permission
  // on left.μ, right.μ }
...
lock (n) {
  reorder n.left.μ after n.μ;
  lock (n.left) {
  reorder n.right.μ after n.left.μ;
  lock (n.right) {...}
} }
```

Approaches lock capabilities, but

- Requires explicit reordering
- Full permissions for reordering loses external references

Fails to exploit that this structure *doesn't need* ordering on children