

Static Lock Capabilities for Deadlock Freedom

Colin S. Gordon Michael D. Ernst Dan Grossman

University of Washington

{csgordon,mernst,djg}@cs.washington.edu

Abstract

We present a technique — lock capabilities — for statically verifying that multithreaded programs with locks will not deadlock. Most previous work on deadlock prevention requires a strict total order on all locks held simultaneously by a thread, but such an invariant often does not hold with fine-grained locking, especially when data-structure mutations change the order locks are acquired. Lock capabilities support idioms that use fine-grained locking, such as mutable binary trees, circular lists, and arrays where each element has a different lock.

Lock capabilities do not enforce a total order and do not prevent external references to data-structure nodes. Instead, the technique reasons about static capabilities, where a thread already holding locks can attempt to acquire another lock only if its capabilities allow it. Acquiring one lock may grant a capability to acquire further locks; in data-structures where heap shape affects safe locking orders, the heap structure can induce the capability-granting relation. Deadlock-freedom follows from ensuring that the capability-granting relation is acyclic. Where necessary, we restrict aliasing with a variant of unique references to allow strong updates to the capability-granting relation, while still allowing other aliases that are used only to acquire locks while holding no locks.

We formalize our technique as a type-and-effect system, demonstrate it handles realistic challenging idioms, and use syntactic techniques (type preservation) to show it soundly prevents deadlock.

Categories and Subject Descriptors D.2.4 [Program Verification]: Correctness proofs; D.3.3 [Language Constructs and Features]: Concurrent programming structures

General Terms Languages, Theory, Verification

Keywords Deadlock, Uniqueness, Concurrency, Capabilities

1. Introduction

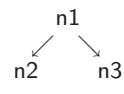
Deadlock occurs when there is a cycle of threads, each blocked waiting for a resource (usually a lock) held by the next thread in the cycle. Deadlock in concurrent software remains a problem despite years of experience in industry and research. State-of-the-art static techniques for preventing deadlock work well for some programs, but sometimes differ greatly from how programmers reason about avoiding deadlock, and they are ill-suited for certain classes of important programs. We propose a technique to address those shortcomings. Our system makes it possible to express locking orders in a more expressive manner, and to verify deadlock freedom

for algorithms not captured by previous work. Our solution also complements the core approach used in most previous work: the two approaches can be combined to yield a yet more expressive system.

1.1 Fine-grained Locking and Deadlock Freedom

A long line of research develops techniques that avoid deadlock for coarse-grained locking, where a lock guards access to an entire data structure. The literature on deadlock freedom for fine-grained locking, where different locks guard different parts of a larger structure, is less developed. No prior technique for static deadlock freedom can verify that the following four threads are deadlock free (which they are) when $n2 == n1.left$ and $n3 == n1.right$:

```
T1 : sync n2 {}
T2 : sync n3 {}
T3 : sync n1 {sync n1.left {sync n1.right {}}}
T4 : sync n1 {sync n1.right {sync n1.left {}}}
```



Most prior techniques either require a total order on the locks acquired [5, 16] (precluding thread 3 or thread 4), or assume strong encapsulation for recursive structures [1] (precluding the interior pointers $n2$ and $n3$ of threads 1 and 2). Verifying deadlock freedom for fine-grained locking becomes even more difficult for mutable data structures where locks may be reordered over time. Another complication is early lock releases (releasing a lock before another held lock that is safe to acquire after the first). Finally, verifying locking orders based on mutable heap structure must also ensure the relevant portion of the heap remains acyclic.

There are important examples similar to the above that are handled by few existing static techniques:

Trees Only a few static approaches [1, 5, 16] can verify deadlock freedom in binary trees whose structure changes over time, such as splay trees (shown in Figure 1 and discussed in Section 4.1).

Array Element Locking While array indices impose a total ordering on array elements (assuming no duplicate entries), verifying that elements were locked according to that order requires either a powerful integer solver or programmer aid in the form of writing explicit branches to acquire locks differently depending on which of multiple indices is larger. We address this example in Section 4.2.

Circular Lists Operating system kernels often use a circular list of running processes. For performance, the list nodes (processes) are locked individually. Consider atomically transferring a resource between processes: this requires locking *multiple* processes *simultaneously*. With a circular list, the traditional approach of requiring all threads to acquire shared objects in a consistent order falls flat because there is no sensible logical ordering on process locks short of resorting to memory addresses. Encapsulation-based techniques also fail because the process locks must reference each other and be directly accessible to all kernel subsystems. We address this example in Section 4.4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

TLDI'12, January 28, 2012, Philadelphia, PA, USA.

Copyright © 2012 ACM 978-1-4503-1120-5/12/01...\$10.00

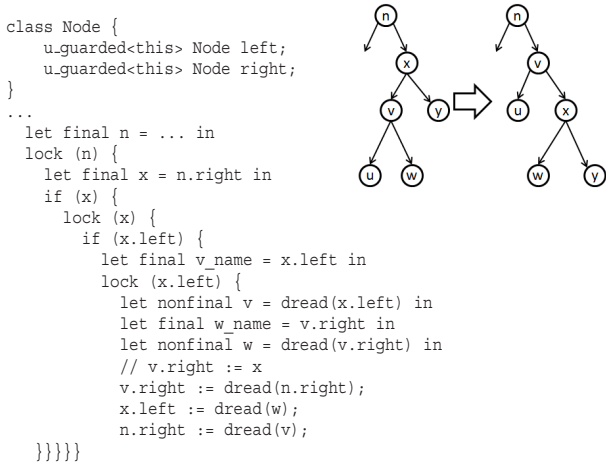


Figure 1. Splay tree rotation in the core language. Assume n refers to the node from which to rotate. $dread$ is a destructive read, and variables may be bound as `final` (i.e. non-assignable) or nonfinal. The extra final variables (x , v_name , w_name) provide names for the type system to track.

1.2 Our Approach

Our technique — *lock capabilities* — handles the examples above and more. The core idea is a simple but expressive locking protocol that can be embedded in a type system or other verification technique. Assuming a tree-shaped partial order on locks, to a first order the locking protocol is as follows:

- A thread that holds no locks may acquire any (single) lock.
- A thread may acquire any immediate successor (in the tree ordering on locks) of a lock it currently holds.

We say that acquiring a lock named x grants the holding thread a unique capability $\langle x \rangle$ as long as the lock is held. We say that x 's children (immediate successors) in the tree-shaped partial order are *guarded by* that capability, and that x grants the capability to acquire its successors. Intuitively, this approach is sound because the fact that this *capability-granting relation* is acyclic ensures that no threads will deadlock with each other by following the order, and using locks for exclusive capability ownership ensures that no threads will deadlock acquiring locks guarded by the same capability.¹

Two features of lock capabilities make them well-suited for fine-grained locking:

Flexible Acquisition Orders Lock capabilities do not require a total order on all locks held simultaneously by a single thread. This is because the thread holding a lock x is the only thread that can lock more than one lock guarded by x 's capability. (Another thread could acquire one such lock as its first lock.) This flexibility lets lock capabilities support examples like fine-grained locking among set elements with no natural ordering among them (e.g. the circular list above): suppose that each element is guarded by the same per-set capability; then threads that must hold multiple elements' locks simultaneously are serialized via the per-set lock, but parallelism with threads that require only one element's lock is still permitted.

Natural Partition Reordering If part of the capability-granting relation can be isolated to one thread, then that thread can safely

¹Reordering locks and early lock releases introduce a subtlety when a thread holds a lock l and an ancestor of l , but not l 's immediate predecessor in the ordering; the necessary restrictions are described in Section 3.5.

Programs	P	$::=$	$\overline{\text{class } e}$
	<i>class</i>	$::=$	<code>class c {<i>field</i>}</code>
	<i>field</i>	$::=$	τf
Paths	p	$::=$	$x \mid p.f$
Expressions	e	$::=$	if $e e e \mid p := e \mid \text{lock } p e$ $\mid \text{dread}(p) \mid \text{spawn } x e \mid \text{null} \mid p$ $\mid \text{let } q x = e \text{ in } e$ $\mid \text{let } u.\text{root } x, \text{ final } y = \text{new } c \text{ in } e$
Final Qualifiers	q	$::=$	nonfinal \mid final
Types	τ	$::=$	αc
Base Types	α	$::=$	$u \mid \text{guardless} \mid \text{borrowed}(x)$
Unique Types	u	$::=$	$u.\text{root} \mid u.\text{guarded}(x)$
		$r, t, x, y, z \in$ Variables	$f \in$ Fields $c \in$ Classes

Figure 2. Core language syntax.

change the relation. In cases where heap structure should dictate the locking order, such as in tree-shaped data structures, reordering follows naturally as long as cycles are not introduced into the capability-granting relation. This can be enforced by using unique references to carry the guard information and strong updates to change a lock's guard. Traditional unique references prevent sharing a structure across threads, so we introduce the concept of partially-unique references. A *u_guarded* reference to an object is the only (unique) reference with the guard in its type. Other references — *guardless* references — may alias the *u_guarded* reference but carry only the class type of their referent, and are therefore only suitable for acquiring a thread's first lock. Mutating the structure through *u_guarded* references naturally expresses the desired changes in the capability-granting relation. For examples like the splay tree in Figure 1, mutations with the *u_guarded* references to nodes also express the changes in the capability-granting relation. We also discuss a way to define the capability-granting relation that is unrelated to heap references (Section 4.4).

This paper presents the lock capabilities verification approach as a static capability type system [20]: if the program type-checks, then the program will not deadlock. To enforce the key invariants of our system, we build on a range of prior work, including work on uniqueness [14] for controlling aliasing, a shallow embedding of graph rewriting to reason about lock ordering cycles and reachability, and work on static race freedom that uses lightweight singleton types [10] to name locks and their associated static capabilities consistently.

Sections 2 and 3 present a type system to verify deadlock freedom statically using lock capabilities. Section 4 illustrates how lock capabilities enable static verification for several challenging examples that can be checked by few if any prior systems: splay tree rotation, locking multiple array elements, and locking multiple nodes of a circular list. Section 5 describes the proof of soundness for the lock capabilities type-and-effect system. Finally, Section 6 compares lock capabilities to prior approaches, and Section 7 concludes.

2. Core Language and Operational Semantics

To explain our work and prove it sound, we define a core language with classes, objects, and fields but — for simplicity — omitting methods (Section 4.3 describes how to add them). Figure 2 presents the syntax. Aside from the types, the language is mostly straightforward, containing conditionals, mutable variables and objects, synchronization on objects (like in Java, every object can serve as a lock), destructive reads, a spawn operation to create a new thread, a constant null value, field dereferences, variable binding, and allocation (with the result bound to two locals — see below). We focus on structured locking in this paper, rather than unstructured with explicit lock and unlock statements. This is not a fundamen-

tal restriction, and extension to unstructured locking is discussed in Section 4.6.

The types include two types of unique references: $u_guarded\langle x \rangle$ references are unique references that associate their referent with a capability and are stored only in fields, and u_root references are unique references whose referents are not associated with any capability and are stored only in variables. $U_guarded$ references are so named because they are the only references that carry the guard of their referent in their type. $dread(p)$ is a destructive read of a path expression, used to atomically set a reference to null and return its old value; this is standard when formalizing systems with unique references. It is possible to convert a $u_guarded$ reference to a u_root reference by destructively reading a field, and to convert the other direction by storing a u_root reference into a $u_guarded$ field. These operations break and create associations, respectively, between objects and guarding capabilities. There are also borrowed references (temporary aliases of $u_guarded$ references), $borrowed\langle x \rangle$, which may only be stored in variables; and regular references with no duplication restriction (called guardless references here because they carry no guard information), which may be stored in fields or variables. Section 3 describes the types in more detail.

The syntax for allocation and thread creation is unusual. Allocation takes the form of a lexical binding that binds two variables to the newly allocated location, one as a mutable unique reference and one as a final (cannot be rebound) variable bound to a guardless reference. The final reference is necessary to name the new object in the type system as soon as it is allocated, since the type system immediately adds assertions about the new allocation; this is not uncommon in type systems with simple singleton types [10, 18, 19]. The unique reference must necessarily be mutable so it can remain useful (be destructively read). The `spawn` operation specifies a single variable to pass (by destructive read) to the new thread’s expression, allowing transfer of a unique reference to the new thread.

Figure 3 presents the operational semantics, including evaluation contexts and syntax extensions for run-time forms. A program state consists of a heap H and a set of threads Ts . The heap maps locations l to objects of the form $\langle c, F, t \rangle$ where c is the class tag, F is a map from field names to values, and t is an optional thread identifier representing which thread, if any, holds the implicit lock associated with that object. Each thread has a unique identifier tid , a map V from variables to values, a list of held locks Ls , and an expression e being reduced. Each heap location has at most one distinguished reference (l^\bullet) that will be typed as $u_guarded$ or u_root . The tags (\bullet) have no runtime effect; they only simplify proving that uniqueness is preserved. The semantics consist of three transition functions:

- $H, Ts \rightarrow H', Ts'$ selects a thread to reduce and reduces it. It also handles spawning a new thread.
- $H, T \rightarrow H', T'$ selects the innermost expression to evaluate.
- $H, tid, V, Ls, e \rightarrow H', V', Ls', e'$ reduces the expression e of thread tid in heap H , environment V , and lock set Ls .

Most of the rules are standard, with a couple minor exceptions, below. We assume an α -renamed program, so no two `let` expressions bind the same variable.

- **E-SPAWN:** The rule for spawning a thread performs a destructive read on a specific variable.
- **E-WVAR, E-DVAR, E-VAR:** We model mutable variables rather than performing binding by substitution. This simplifies the semantics for destructive reads. We use the notation $Dup(v)$ to preserve uniqueness of tagged value; it ensures there is at most one l^\bullet for any l in the heap’s domain ($Dup(l^\bullet) = l$).

- **E-*LOCK:** The semantics support recursive lock acquisition (repeated acquisition by the holding thread). When an object’s implicit lock is not held, the owner field of its heap value is `None`. When the owner field is `Some(tid)`, the thread holding the lock has identifier tid (E-LOCK). A thread can acquire a lock it already owns (E-RELOCK). When releasing the lock, the rules differentiate a recursively acquired lock (E-RECUNLOCK) from a normal lock (E-UNLOCK) by checking if the lock l being released appears in the lockset after removing the most recently-acquired lock.
- **E-NEW:** Allocation binds two variables, as mentioned above. One copy of the value is unique, while the other is not. This is not the standard notion of a unique reference, but will be explained in full in Section 3.2.
- **E-DVAR, E-DFIELD:** The semantics provide destructive reads that atomically set a variable or field to null and return the original value. This is common for semantics with unique references, as such an operation avoids duplicating a reference.

3. Core Typing Ideas

This section describes the main ideas of the type system. The type system preserves two primary properties:

- If a thread holds no locks, it may acquire any lock. Otherwise a thread may acquire only locks for which it holds capabilities.
- The capability-granting relation (the relation determining which other locks each lock grants a capability to acquire) is acyclic.

The main typing judgement is $Y; \Gamma; L \vdash e : \tau; Y'$. Here Y tracks trees and subtrees in the capability-granting relation, and which (sub)trees are mutually disjoint. When checking code that modifies the capability-granting relation, Y is consulted to verify that a modification will not introduce a cycle. Because the relation can change, Y' is the new capability-granting relation after executing e . Treatment of Y and Y' is detailed in Section 3.6. Γ maps variables to their types, with a qualifier describing variables as final (i.e., cannot be rebound) or non-final. L is a list of final variables, each aliased to a lock held at that program point during execution, making it the static counterpart to Ls in the operational semantics. L also doubles as the set of capabilities held at a program point; a static check that a capability is possessed is a check for membership in L . Informally, in a dynamic state where the assertions in Y hold, Γ provides accurate types for variables, and the final variables in L correspond to the runtime locks held, executing the expression e will produce a value of type τ and change the capability-granting relation so the assertions in Y' hold (or get stuck trying to dereference null).

The remainder of this section explains the main typing ideas. Section 3.1 elaborates on the role of capabilities. Section 3.2 explains our use of unique references, and our extension to *partial uniqueness*. Section 3.3 explains the type system’s use of an external must-alias analysis. Section 3.4 explains how we ensure that side effects in one thread cannot invalidate typing information in another thread. Section 3.5 explains the “orphaned lock” problem introduced by early lock releases and changes to the capability-granting relation. Section 3.6 explains how the type system keeps the capability-granting relation acyclic, and Section 3.7 introduces the typing rules.

3.1 Static Capabilities

A capability is acquired by acquiring a lock. For example, if a thread acquires a lock that is must-aliased with the final variable x , then while that lock is held the thread possesses the capability $\langle x \rangle$. This capability permits acquiring locks guarded by $\langle x \rangle$. Note that x is not guarded by $\langle x \rangle$: that would be a cycle. We sometimes refer to

<table style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px;">Locations</td><td style="padding: 2px;">l</td></tr> <tr><td style="padding: 2px;">Field Maps</td><td style="padding: 2px;">$F : \text{Field} \mapsto \text{Value}$</td></tr> <tr><td style="padding: 2px;">Values</td><td style="padding: 2px;">$v ::= l \mid l^* \mid \text{null}$</td></tr> <tr><td style="padding: 2px;">Program States</td><td style="padding: 2px;">$S : \text{Heap} * (\text{Thread set})$</td></tr> <tr><td style="padding: 2px;">Environment</td><td style="padding: 2px;">$V : \text{Variable} \mapsto \text{Value}$</td></tr> <tr><td style="padding: 2px;">Paths</td><td style="padding: 2px;">$p ::= \dots \mid l$</td></tr> </table>	Locations	l	Field Maps	$F : \text{Field} \mapsto \text{Value}$	Values	$v ::= l \mid l^* \mid \text{null}$	Program States	$S : \text{Heap} * (\text{Thread set})$	Environment	$V : \text{Variable} \mapsto \text{Value}$	Paths	$p ::= \dots \mid l$		<table style="width: 100%; border-collapse: collapse;"> <tr><td style="padding: 2px;">Heaps</td><td style="padding: 2px;">H</td><td style="padding: 2px;">$:$</td><td style="padding: 2px;">$\text{Location} \mapsto \text{Class} * (\text{Field Map}) * (\text{ThreadID Option})$</td></tr> <tr><td style="padding: 2px;">Threads</td><td style="padding: 2px;">T</td><td style="padding: 2px;">$:$</td><td style="padding: 2px;">$\text{ThreadID} * \text{Environment} * (\text{Held Locks}) * \text{Expression}$</td></tr> <tr><td style="padding: 2px;">Thread Sets</td><td style="padding: 2px;">Ts</td><td style="padding: 2px;">$:$</td><td style="padding: 2px;">Thread set</td></tr> <tr><td style="padding: 2px;">Held Locks</td><td style="padding: 2px;">Ls</td><td style="padding: 2px;">$:$</td><td style="padding: 2px;">Location list</td></tr> <tr><td style="padding: 2px;">Expressions</td><td style="padding: 2px;">e</td><td style="padding: 2px;">$::=$</td><td style="padding: 2px;">$\dots \mid \text{withlock } l \ e \mid l \mid l^*$</td></tr> </table>	Heaps	H	$:$	$\text{Location} \mapsto \text{Class} * (\text{Field Map}) * (\text{ThreadID Option})$	Threads	T	$:$	$\text{ThreadID} * \text{Environment} * (\text{Held Locks}) * \text{Expression}$	Thread Sets	Ts	$:$	Thread set	Held Locks	Ls	$:$	Location list	Expressions	e	$::=$	$\dots \mid \text{withlock } l \ e \mid l \mid l^*$
Locations	l																																	
Field Maps	$F : \text{Field} \mapsto \text{Value}$																																	
Values	$v ::= l \mid l^* \mid \text{null}$																																	
Program States	$S : \text{Heap} * (\text{Thread set})$																																	
Environment	$V : \text{Variable} \mapsto \text{Value}$																																	
Paths	$p ::= \dots \mid l$																																	
Heaps	H	$:$	$\text{Location} \mapsto \text{Class} * (\text{Field Map}) * (\text{ThreadID Option})$																															
Threads	T	$:$	$\text{ThreadID} * \text{Environment} * (\text{Held Locks}) * \text{Expression}$																															
Thread Sets	Ts	$:$	Thread set																															
Held Locks	Ls	$:$	Location list																															
Expressions	e	$::=$	$\dots \mid \text{withlock } l \ e \mid l \mid l^*$																															
<p style="margin: 0;">Evaluation Contexts $E ::= [\cdot] \mid \text{if } E \ e \ e \mid E.f \mid E.f := e \mid l.f := E \mid x := E \mid \text{lock } E \ e \mid \text{dread}(E.f) \mid \text{let } q \ x = E \ \text{in } e \mid \text{withlock } l \ E$</p>																																		
$H, Ts \rightarrow H', Ts'$	<p style="margin: 0;">E-THREAD $\frac{H, T \rightarrow H', T'}{H, \{T\} \cup Ts \rightarrow H', \{T'\} \cup Ts}$</p>	<p style="margin: 0;">E-SPAWN $\frac{\text{tid}' \ \text{fresh} \quad V' = V[x \mapsto \text{null}] \quad V_{\text{new}} = \{x \mapsto V(x)\}}{H, \{(tid', V, Ls, E[\text{spawn } x \ e])\} \cup Ts \rightarrow H, \{(tid', V', Ls, E[\text{null}]), (tid', V_{\text{new}}, [], e)\} \cup Ts}$</p>																																
$H, T \rightarrow H', T'$	<p style="margin: 0;">E-CONTEXT $\frac{H, tid, V, Ls, e \rightarrow H', V', Ls', e'}{H, (tid, V, Ls, E[e]) \rightarrow H', (tid, V', Ls', E[e'])}$</p>																																	
$H, tid, V, Ls, e \rightarrow H', V', Ls', e'$	<p style="margin: 0;">E-IF-TRUE $\frac{}{H, tid, V, Ls, \text{if } l^{[*]} \ e_1 \ e_2 \rightarrow H, V, Ls, e_1}$</p>	<p style="margin: 0;">E-IF-FALSE $\frac{}{H, tid, V, Ls, \text{if } \text{null} \ e_1 \ e_2 \rightarrow H, V, Ls, e_2}$</p>																																
<p style="margin: 0;">E-WVAR $\frac{}{H, tid, V, Ls, x := v \rightarrow H, V[x \mapsto v], Ls, \text{Dup}(v)}$</p>	<p style="margin: 0;">E-WFIELD $\frac{}{H, tid, V, Ls, l.f := v \rightarrow H[l.f \mapsto v], V, Ls, \text{Dup}(v)}$</p>																																	
<p style="margin: 0;">E-LOCK $\frac{H(l) = \langle c, F, \text{None} \rangle}{H, tid, V, Ls, \text{lock } l \ e \rightarrow H[l \mapsto \langle c, F, \text{Some}(tid) \rangle], V, l :: Ls, \text{withlock } l \ e}$</p>	<p style="margin: 0;">E-UNLOCK $\frac{l \notin Ls \quad H(l) = \langle c, F, \text{Some}(tid) \rangle}{H, tid, V, l :: Ls, \text{withlock } l \ v \rightarrow H[l \mapsto \langle c, F, \text{None} \rangle], V, Ls, v}$</p>																																	
<p style="margin: 0;">E-RELOCK $\frac{H(l) = \langle c, F, \text{Some}(tid) \rangle}{H, tid, V, Ls, \text{lock } l \ e \rightarrow H, V, l :: Ls, \text{withlock } l \ e}$</p>	<p style="margin: 0;">E-RECUNLOCK $\frac{l \in Ls \quad H(l) = \langle c, F, \text{Some}(tid) \rangle}{H, tid, V, l :: Ls, \text{withlock } l \ v \rightarrow H, V, Ls, v}$</p>																																	
<p style="margin: 0;">E-NEW $\frac{\text{class } c\{\tau_1 \ f_1 \dots \tau_n \ f_n\} \in P \quad F = \{f_1 \mapsto \text{null}, \dots, f_n \mapsto \text{null}\} \quad l \notin \text{Dom}(H)}{H, tid, V, Ls, \text{let } u.\text{root } x, \text{ final } y = \text{new } c \ \text{in } e \rightarrow H[l \mapsto \langle c, F, \text{None} \rangle], V[x \mapsto l^*, y \mapsto l], Ls, e}$</p>	<p style="margin: 0;">E-VAR $\frac{}{H, tid, V, Ls, x \rightarrow H, V, Ls, \text{Dup}(V(x))}$</p>																																	
<p style="margin: 0;">E-FIELD $\frac{}{H, tid, V, Ls, l.f \rightarrow H, V, Ls, \text{Dup}(H(l)(f))}$</p>	<p style="margin: 0;">E-DVAR $\frac{}{H, tid, V, Ls, \text{dread}(x) \rightarrow H, V[x \mapsto \text{null}], Ls, V(x)}$</p>																																	
<p style="margin: 0;">E-DFIELD $\frac{}{H, tid, V, Ls, \text{dread}(l.f) \rightarrow H[l.f \mapsto \text{null}], V, Ls, H(l)(f)}$</p>	<p style="margin: 0;">E-LET $\frac{}{H, tid, V, Ls, \text{let } q \ x = v \ \text{in } e \rightarrow H, V[x \mapsto v], Ls, e}$</p>																																	

We use a shorthand $H[l.f \mapsto v] \equiv H[l \mapsto \langle c, F[f \mapsto v], tid \rangle]$ where $H(l) = \langle c, F, tid \rangle$. For generating aliases of unique values, we use the function $\text{Dup}(v) = \text{if } (v = l^*) \ \text{then } l \ \text{else } v$.

Figure 3. Operational Semantics

a set of locks guarded by the same capability as being in the same *lock group*. Capabilities exist only in the type system; capabilities and lock groups have no runtime representation.

To give the type system stable names to refer to held locks, the type system enforces that all locks taken are must-aliased to final variables. This is common practice in other systems with variants of singleton types, including some race-freedom work [10, 18, 19].

L is a static representation of the list Ls of dynamically held locks (from the semantics in Figure 3). Replacing each final variable $x \in L$ with the location it maps to in the dynamic local environment $V(x)$ produces exactly Ls when typing the next redex to be reduced. Ignoring order, $(\exists x \in L. V(x) = l) \Leftrightarrow l \in Ls$. The type rules for lock acquisition extend L in the same way the evaluation rules extend Ls . In the core language presented here, L exactly represents the dynamic lock set, because there are no methods or loops. However, neither the type judgements nor the soundness proof relies on this fact.

We must ensure that a lock statement acquires a lock guarded by the capability of a held lock (or that no locks are held). Checking that an expression being locked is in the lock group of a capability the thread possesses appears in the type rules as a check for membership in L . If the expression p types as a path to a lock guarded by $\langle x \rangle$, then if $x \in L$ it is statically safe to lock p . In the type system (Fig. 6), this membership check corresponds to the hypothesis $y \in L$ where the target lock's type is borrowed $\langle y \rangle \ c$ in rule T-LOCK-N.

3.2 Uniqueness, Partial Uniqueness, and Borrowing

Our system statically enforces that at most one lock grants the capability to acquire each lock. To ensure this while still permitting updates to the capability-granting relation, we use a special form

of unique references. Syntactically, $\text{u_guarded}\langle x \rangle \ c$ is a unique reference to an object of class c guarded by the capability $\langle x \rangle$. In our core language's source syntax, a field's type may only refer to the capability $\langle \text{this} \rangle$ because this is the only capability name in scope in that context; Section 4.4 describes an extension for using other objects' capabilities. $\text{u_root } c$ is a unique reference to an object of class c , but has no guard, and is therefore a root in the capability-granting relation.

As in many systems with unique types, we use destructive reads to preserve uniqueness. We also use writes and destructive reads on u_guarded references (as in $\text{u_guarded}\langle x \rangle \ c$) to perform strong updates to the guard portion $\langle x \rangle$ of an object's type: destructively reading a u_guarded field removes the referent from its previous parent's lock group, and storing a u_root reference into a u_guarded field moves the referent into its new parent's lock group. Object fields may not be u_root references, and local variables may not be u_guarded references (allowing both qualifiers in both places is not conceptually difficult, but preventing it here simplifies tracking the capability-granting relation — see Section 3.6). For example:

```

// assume x is final, with u_guarded field f
let nonfinal y = dread(x.f) in
// y is u_root, x.f holds null
x.f := dread(y)
// x.f contents again u_guarded<x>, y holds null

```

u_guarded references are necessary only for controlling the capability-granting relation. Most references are normal, and their types do not need to carry the referent's guard. We call these *guardless references* (with type $\text{guardless } c$). Note that we allow guardless references to objects for which there is also one unique

reference. This is sound because with a guardless reference only reads, writes, and acquiring a first lock are permitted. In a typical program, most references would be guardless.

“Borrowing” refers to allowing the use of a unique reference without consuming it: making a temporary local copy without being required to destroy the original. Borrowing lets the type system use an object’s lock group (i.e., to check that a thread possesses the capability for a lock being newly acquired) without requiring the program to modify the original unique reference. Systems without borrowing end up with the awkward idiom of explicitly threading unique references through computations just to restore the unique reference back into its original location.

When a regular read is performed on a unique field, it is treated as a borrowing read. A metafunction on types, $\text{Alias}(\tau)$, computes the result type of a borrowing (aliasing) read on a field or variable of type τ . See the definition in Figure 5. A regular non-destructive read on a $\text{u_guarded}(x)$ c has type $\text{Alias}(\text{u_guarded}(x) c) = \text{borrowed}(x) c$. This type represents a non-unique reference to an object with the same class as the u_guarded reference, carrying the same lock group information. A regular read on a u_root variable simply returns a guardless reference ($\text{Alias}(\text{u_root } c) = \text{guardless } c$) because there is no guard to borrow. Note that the use of $\text{Alias}(\tau)$ roughly corresponds to uses of $\text{Dup}(v)$ in the operational semantics (Figure 3), because borrowing occurs when a unique value may be aliased and $\text{Dup}(v)$ preserves uniqueness of tagging by producing an untagged copy of a tagged value when an aliasing read duplicates a reference. The discrepancies are places where $\text{Alias}(\tau)$ can be omitted because there is enough information to predict its result. For example, the type system has separate rules for writes to guardless fields and u_guarded fields. In the former case, the write’s result will always be guardless. $\text{Alias}(\text{guardless } c) = \text{guardless } c$ so we simplify the result type.

A unique reference’s lock group information is always valid (u_root references hold valid guard information; they imply the referent has no guard), but a borrowed reference has accurate lock group information only if the corresponding unique reference exists. A borrowed reference could end up with stale lock group information if the u_guarded reference it is borrowed from is destructively read (or overwritten) since that would change the capability guarding the referent. The lock group information of a borrowed reference r with type $\text{borrowed}(x) c$ is valid only if there is a static must-alias p of the reference r at the current program point that holds a unique reference in the same lock group as the borrowed type. To check that the borrowed reference p ’s lock group information matches the lock group x of the corresponding u_guarded reference, we use the metafunction $\text{ValidCap}(\Gamma, L, p, x)$ (for “valid capability”) when reading variables or acquiring locks, to prevent use of stale lock group information. If a borrowed reference is no longer valid, that reference can no longer be used. Note that unlike the traditional uses of borrowing which are concerned with avoiding persistent duplication of a unique reference, we use borrowing to temporarily use the guard portion of a type, and avoid persistently duplicating that typing assumption. As an example:

```
// assume x is final, with u_guarded field f
let nonfinal y = x.f in // y:borrowed<x>
// y==x.f so y's guard is valid
let nonfinal a = dread(x.f) in // a:u_root
// y!=x.f, so y's guard is now invalid and
// uses of y will not type check
let nonfinal b = a in
// b:guardless, since a:u_root
```

3.3 Aliasing Information

We need must-aliasing information for three reasons, each discussed in detail in another section:

1. We need aliasing information to check validity of a borrowed reference’s lock group information (Section 3.2).
2. We need aliasing information to retain expressiveness in the face of path mutation. We need it to associate objects accessed through paths with static lock names (Section 3.1).
3. One of the core principles for soundness of our approach is that there is no cycle in the capability-granting relation. Aliasing information allows the type system to reason about the safety of field updates that implicitly affect the capability-granting relation, such as an update performed through one variable (as in $x.f := e$) when information about the capability granting relation is tracked in terms of some syntactically unrelated local root y where $y.g$ is aliased to x (Section 3.6).

We assume a sound must-alias analysis is available, defined outside our system. An alternative would be for the type system to track aliasing directly or to adapt a system like alias types [18, 19]. Formalizing such must-aliasing in the type system would add significant complexity to our formalization that is not directly relevant to the core ideas of lock capabilities. Using an external analysis simplifies the presentation and demonstrates that the system is sound for any sound must-alias analysis, rather than just a particular analysis encoded in type rules.

The query $\text{MustAlias}(\cdot, p)$ returns a set of paths that are aliases of p at the current program point (\cdot). We assume, as is common in the pointer analysis literature, that each runtime expression is implicitly labeled with the source expression location θ it came from, and that the operational semantics propagate these labels appropriately. Passing \cdot to the alias analysis is equivalent to looking up the label for the program point immediately after evaluation of the expression being checked: $\text{MustAlias}(\cdot, p) \equiv \text{MustAlias}(\theta, p_\theta)$. The must-alias analysis is queried throughout the type judgements, both explicitly, and via some of the macros in Figure 5.

3.4 Heap Partitioning Between Threads

Because individual threads are type checked separately, a reduction in one thread must not invalidate typing information or must-alias results in another thread.

The simplest sound way to ensure that one thread will not perform a field update that invalidates another thread’s assumptions is to enforce data race freedom. For simplicity, we maintain an even stronger property: that threads may make assumptions only about disjoint portions of the heap at run time. We maintain a simple invariant, that a thread uses the types or aliasing information for an object’s fields only when the thread has locked that object. Race freedom is reflected in our type system by two invariants:

- **Disjoint Lock Sets:** This is a standard invariant for any system dealing with concurrency, which basically means that no lock is held simultaneously by more than one thread (which is the very purpose of mutual exclusion locks).
- **Race Free Field Access:** Thread typing may use field information and query must-alias information only for fields of objects that are in the current thread’s lock set. This is enforced with uses of $\text{RaceFreePath}(\Gamma, L, p)$ in the typing rules.

Once we can ensure the invariants above, ensuring that a write to the heap does not invalidate other threads’ assumptions is straightforward. To ensure that no lock group information is invalidated by destructive reads changing the lock group of u_guarded references, we also maintain the invariant that for each dynamic location, at most one thread’s typing context associates it with a lock group.

```

... // initially a->b->c
synchronized(n) { // a
  synchronized(n.next) { // b
    synchronized(n.next.next) { // c
      Node tmp = n.next;
      n.next = tmp.next;
      tmp.next = n.next.next;
      n.next.next = tmp;
    } // a->c->b, only a and b held
    synchronized(n.next) {
      // Potential Deadlock!
    }
  }
}

```

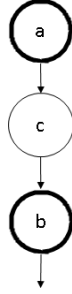


Figure 4. Java code demonstrating an “orphaned lock” (lock b) in the context of a singly-linked list. The diagram on the right shows the heap (and capability-granting) structure after the first lock release. The nodes with dark borders are locked at that point by the thread running the code above. A second thread could deadlock with this code if it acquires lock c , then tries to lock c .next at the same time as the code above tries to re-acquire a .next.

Note that while we enforce data race freedom, it is not strictly necessary. Allowing data races on guardless reference fields does not risk deadlock; a thread may only lock a guardless reference when it holds no other locks, and the use of must-aliasing information can be adjusted to allow interference on those fields. Unsynchronized *reads* of *u_guarded* references can be made sound if any unsynchronized read produces a guardless reference (synchronized reads may still return borrowed references). What truly must be race free is the use of the guard information encoded in *u_guarded* fields. Our enforcement of *u_guarded* data race freedom is only to simplify presentation.

3.5 Orphaned Locks

Without changes to the capability-granting relation, any set of locks a thread holds at one time will span some contiguous subtree of the capability-granting relation (because we use structured locking without an explicit unlock). In such cases, ensuring that the granting relation remains acyclic is sufficient for safety, as will be discussed in Section 3.6. However, with changes in the capability-granting relation through destructive reads and stores of unique references, a thread can hold locks in multiple disconnected subtrees of the capability-granting relation. Once some set of locks are released after rearranging contiguous subtrees, it is possible for a thread to hold a non-contiguous set of locks in a single subtree.

If a thread holds two locks, one a capability-grant descendant of the other, but not all locks in between, it is not safe for the thread to acquire intermediate locks it does not hold, even though it holds the capability required. Figure 4 shows that treating this scenario naïvely could permit deadlock. If a grants the capability to acquire c , which is not locked by thread A but (transitively) grants the capability to acquire b , and thread A holds the lock a and its capability-granting descendant b , then acquiring c could deadlock. A second thread B could acquire the lock c as its first lock, use c ’s capability to attempt to acquire b , and block waiting for thread A . Then A would block waiting for B when attempting to acquire c .

An orphaned lock is a held lock (other than the first one acquired) for which the thread no longer holds the capability that would allow it to be acquired. In our core language, this manifests at the type level by being unable to locate the *u_guarded* reference to a held lock other than the first acquired (in Figure 4, b is orphaned because the thread lacks access to the *u_guarded* reference for b). Fortunately, a solution is simple: while any lock other than the first one acquired is orphaned, do not permit any further lock acquisitions. This restriction is imposed in the type system by the type rule for acquiring locks after the first. In Figure 4, this means that

after the lock b becomes orphaned, until the lock b is released, the type system will not permit further lock acquisition by this thread.

More precise tracking of where the orphaned lock was would enable acquisition of other safe locks, but we have not found it necessary for our core language with structured locking. If lock capabilities were adapted for a language with explicit lock and unlock statements, such precise tracking to permit the use of safe capabilities (like b ’s in Figure 4) would be highly desirable. Such an extension would allow idioms such as hand-over-hand locking, or even following two separate hand-over-hand paths through the capability-granting relation. Note that in such a system, the orphaned lock problem would arise even without reordering because a thread could explicitly release a parent before a child.

3.6 Tree Reachability Assertions

Soundly preventing deadlock requires that the capability-granting relation remains acyclic. Because each lock can be guarded by at most one capability, the relation is not only acyclic but forms a forest. With this insight, we can create a set of simple rules based on rewriting a forest to verify that forestness is preserved by updates to the capability-granting relation.

Υ is a local view of a part of the global capability-granting relation, which in the formal system is embedded in the heap’s *u_guarded* reference edges. Extensions with capability-grants unrelated to heap structure are considered in Sections 4.4 and 4.5. The type system treats destructive reads of *u_guarded* references as edge removals. Similarly, storing a *u_guarded* reference is equivalent to adding an edge. More detail on Υ appears in a technical report [13].

Υ has three types of assertions about capability-granting trees:

- $x||y$ means the referent of x is not reachable (in the capability-granting relation) from the referent of y and vice versa. This assertion is introduced by destructive reads and allocations. Note that we consider $x||y \equiv y||x$, so $x||y \in \{y||x\}$.
- $\text{root}(y)$ means that there is no incoming edge (in the capability-granting relation) to y ’s referent. This is the result of either destructively reading a *u_guarded* reference, or allocating a new object. No two roots in Υ may alias each other.
- $\text{subtree}(x)$ is a weaker substitute for $\text{root}(x)$ when it is unknown whether there is a *u_guarded* reference to x in the heap, as is often the case when a thread acquires its first lock via a guardless reference. It acts as an anchor to keep Υ well-formed. For every $x||y \in \Upsilon$, either $\text{subtree}(x) \in \Upsilon$ or $\text{root}(x) \in \Upsilon$, and similarly for y . There is at most one subtree assertion in Υ , and it is permitted to alias a root variable depending on the strength of the must-alias analysis.

The typing rules use a judgement $\Upsilon; \Gamma; L \vdash x \mapsto p$ to determine that x is the (subtree) root of the capability-granting tree containing p (Figure 5). This judgement is used in several typing rules to select a root relative to which to add disjointness assumptions. Because a root may alias a subtree, it is unsafe to add disjointness assertions for some root relative to all other roots and subtrees. Doing so could duplicate a disjointness assertion, allowing code to assume and violate the same assertion twice, and create a cycle of *u_guarded* references. So when checking allocation the type system picks a single root or subtree, and adds disjointness assertions relative only to that root and things known to be disjoint from it.

3.7 Formal Type Rules

This section presents the typing rules for our core language. To clarify the presentation, hypotheses related to different issues are shaded differently. There are shadings corresponding to:

- Trees and acyclicity of the capability-granting relation

$$\begin{aligned}
\text{Alias}(\tau) &= \begin{cases} \text{borrowed}(x) \ c & \text{if } \tau = \text{u_guarded}(x) \ c \\ \text{guardless } c & \text{if } \tau = \text{u_root } c \\ \tau & \text{otherwise} \end{cases} \\
\text{FinalAlias}(\Gamma, p, x) &\equiv x \in \text{MustAlias}(\cdot, p) \wedge \Gamma(x) = \text{final } \tau_x \\
\text{LockedFinalAlias}(\Gamma, L, p, x) &\equiv \text{FinalAlias}(\Gamma, p, x) \wedge x \in L \\
\text{RaceFreePath}(\Gamma, L, p) &= \begin{cases} \text{true} & \text{if } p = l \\ \text{true} & \text{if } p = x \\ \text{true} & \text{if } \left(\begin{array}{l} p = p'.f \wedge \\ \text{RaceFreePath}(\Gamma, L, p') \wedge \\ \exists x. \text{LockedFinalAlias}(\Gamma, L, p', x) \end{array} \right) \\ \text{false} & \text{otherwise} \end{cases} \\
\text{FieldAccess}(\Upsilon, \Gamma, L, p, f) &= \tau \text{ if} \\
&\Upsilon; \Gamma; L \vdash p : \tau_1; \Upsilon \wedge \text{RaceFreePath}(\Gamma, L, p) \wedge \\
&\text{LockedFinalAlias}(\Gamma, L, p, x) \wedge P \vdash \tau_2 \ f \in \text{Fields}(\text{Class}(\tau_1)) \wedge \\
&\tau = \tau_2[x/\text{this}] \\
\text{ValidCap}(\Gamma, L, p, x) &= \exists p'. \left(\begin{array}{l} p' \in \text{MustAlias}(\cdot, p) \wedge \\ \Gamma; L \vdash p' : \text{u_guarded}(x) \ c \end{array} \right) \\
\text{NewSubtrees}(L, \Upsilon, x) &= \\
&\text{if } (L \neq [] \vee (\exists t. \text{root}(t) \in \Upsilon \wedge t \in \text{MustAlias}(\cdot, x))) \\
&\text{then } \emptyset \\
&\text{else } \{\text{subtree}(x)\}
\end{aligned}$$

$$\begin{array}{c}
\boxed{\Upsilon; \Gamma; L \vdash x \mapsto p} \quad \frac{\text{ROOT-ROOT} \quad \text{root}(x) \in \Upsilon}{\Upsilon; \Gamma; L \vdash x \mapsto x} \quad \frac{\text{ROOT-SUBTREE} \quad \text{subtree}(x) \in \Upsilon}{\Upsilon; \Gamma; L \vdash x \mapsto x} \\
\text{ROOT-ALIAS} \quad \frac{\Upsilon; \Gamma; L \vdash x \mapsto p' \quad p' \in \text{MustAlias}(\cdot, p)}{\Upsilon; \Gamma; L \vdash x \mapsto p} \\
\text{ROOT-FIELD} \quad \frac{\Upsilon; \Gamma; L \vdash x \mapsto p \quad \Gamma; L \vdash p.f : \text{u_guarded}(y) \ c}{\Upsilon; \Gamma; L \vdash x \mapsto p.f} \\
\boxed{\Gamma; L \vdash p : \tau} \quad \text{T-VAR-STORAGE} \quad \frac{\Gamma(x) = q \ \tau}{\Gamma; L \vdash x : \tau} \\
\text{T-FIELD-STORAGE} \quad \frac{P \vdash \tau_2 \ f \in \text{Fields}(\text{Class}(\tau_1)) \quad \Gamma; L \vdash p : \tau_1 \quad \text{LockedFinalAlias}(\Gamma, L, p, x)}{\Gamma; L \vdash p.f : \tau_2[x/\text{this}]} \\
\text{T-PROGRAM} \\
\frac{\text{FieldsOnce}(\overline{\text{class}}) \quad \text{ClassesOnce}(\overline{\text{class}}) \quad \forall c \in \overline{\text{class}}. \text{class} \vdash c \quad \emptyset; \emptyset; \emptyset \vdash e : \tau; \Upsilon'}{\vdash \text{class } e} \\
\text{T-CLASS} \quad \frac{\forall \tau \ f \in \text{Fields}(c). \overline{\text{class}} \vdash \tau \ f}{\text{class} \vdash c} \quad \text{T-VALIDGUARDLESSFIELD} \quad \frac{c \in \overline{\text{class}}}{\text{class} \vdash \text{guardless } c \ f} \\
\text{T-VALIDU_GUARDEDFIELD} \quad \frac{c \in \overline{\text{class}}}{\text{class} \vdash \text{u_guarded}(\text{this}) \ c \ f}
\end{array}$$

Figure 5. Supporting judgements and program typing

- **Safe lock acquisition** — acquiring only locks for which the thread holds the capability, and not acquiring locks while holding orphaned locks.
- **Unshaded** — basic treatment of unique references (borrowing, destructive reads, writes), and standard features such as allocation and binding, and flow-sensitive propagation of Υ .

On a first reading, the reader may benefit from ignoring **tree** hypotheses, and simply assuming the capability-granting relation is kept acyclic. The complete system, including all shaded hypotheses, ensures deadlock freedom and incidentally prevents simultaneous access to fields (as discussed in Section 3.4).

Figure 5 presents supporting judgements and auxiliary functions used in the main rules. This figure defines program typing, the relation $\Upsilon; \Gamma; L \vdash x \mapsto p$ to decide capability-granting roots, the storage typing relation $\Gamma; L \vdash p : \tau$ to decide the type of value stored in a path (rather than typing the evaluation of reading that path’s

value), and functions for typing borrowing read results, checking that paths are race free, valid capability checks, and a macro to make field accessing rules more readable. Figure 6 gives the source typing rules. The rules for typing run-time expressions appear in Section 5.

We now present details for selected rules from Figure 6.

- **T-VAR** is straightforward, accounting for borrowing via the $\text{Alias}(\tau)$ metafunction (Figure 5), but also a variable typed as borrowed can be read only if the borrowed reference is valid (must-aliased to a u_guarded reference with the same guard).
- **T-WVAR** is mostly standard, except for borrowing the type of the value stored.
- **T-DVAR** is a mostly standard destructive read. Because a destructive read destroys the old value, the return type of a destructive read is a unique reference, not a borrowed reference.
- **T-FIELD** is the most basic field typing, which performs a normal (borrowing) read of a field. $\text{FieldAccess}(\dots)$ (Figure 5) returns the type of field contents stored at the end of a path, also checking that the path is race-free, and that there is a final alias to the last object accessed by reducing the path expression, which provides a name for translating a $\text{u_guarded}(\text{this}) \ c$ field declaration for the current context. It ensures that evaluating the path will not change the capability-granting relation.
- **T-WFIELD-U_GUARDED**, after checking the field access, checks for a final variable y aliased to the expression being stored (MustAlias is defined only for paths and destructive reads of paths). y is used in the last three hypotheses to check whether adding a capability grant from x to y will preserve capability-granting acyclicity (see Section 3.6). All assertions about y are removed from the resulting set of tree assertions, since it would no longer be a root, and any disjointness assertions about it either no longer hold or are redundant with disjointness assertions about the tree it is being added to.
- **T-DFIELD** handles destructive reads of u_guarded references. It is similar to **T-FIELD**, but also checks that the reference being removed is aliased to a final variable t that can be used for tree assertions in the resulting Υ' , which is enriched with assertions that anything disjoint from the original tree r is also disjoint from the tree rooted at t . This rule produces a u_root , rather than u_guarded , reference because the referent is no longer in any lock group.
- **T-NEW** binds final and unique variables to the newly-allocated object. The body of the statement is checked in an environment extended with a final variable y and non-final u_guarded reference x (which are initially aliased), and with tree assertions that nothing is reachable from the newly-allocated object. Only assertions relative to one known root are added, because it is possible to have a root and subtree in Υ aliased to each other, and adding disjointness assertions for both is unsound (as discussed in Section 3.6 and in more detail in our technical report [13]).
- **T-SPAWN** is unusual largely because our core language binds variables in a local environment rather than by substitution. The corresponding evaluation rule destructively reads the variable x in the local context (to preserve uniqueness if x is unique, still safe otherwise), and carries the old dynamic binding over to the new thread as a guardless reference. No tree assertions are removed, because the static must alias analysis may be too weak to find the correct final variable to remove from Υ . Consequently the spawned thread cannot violate disjointness assumptions the parent thread may have about the passed value.

$\Upsilon := \{\text{root}(\text{Variable}) \mid \text{subtree}(\text{Variable}) \mid \text{Variable} \mid \text{Variable}\}$ set	$\Gamma : \text{Variable} \mapsto \text{Final Qualifier} * \text{Type}$	$L : \text{Variable list}$
$\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$	$\text{T-IF} \frac{\Upsilon; \Gamma; L \vdash e_1 : \tau_1; \Upsilon_1 \quad \Upsilon_1; \Gamma; L \vdash e_2 : \tau; \Upsilon_2 \quad \Upsilon_1; \Gamma; L \vdash e_3 : \tau; \Upsilon_3}{\Upsilon; \Gamma; L \vdash \text{if } e_1 \ e_2 \ e_3 : \tau; \Upsilon_2 \cap \Upsilon_3}$	$\text{T-NULL} \frac{}{\Upsilon; \Gamma; L \vdash \text{null} : \text{guardless } c; \Upsilon'}$
$\text{T-VAR} \frac{\Gamma(x) = q \ \tau \quad \tau' = \text{Alias}(\tau) \quad \forall y \forall c. \tau' = \text{borrowed}(y) \ c \Rightarrow \text{ValidCap}(\Gamma, L, x, y)}{\Upsilon; \Gamma; L \vdash x : \tau'; \Upsilon}$	$\text{T-WVAR} \frac{\Gamma(x) = \text{nonfinal } \tau \quad \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'}{\Upsilon; \Gamma; L \vdash x := e : \text{Alias}(\tau); \Upsilon'}$	
$\text{T-DVAR} \frac{\Gamma(x) = \text{nonfinal } \text{u_root } c \quad \text{FinalAlias}(\Gamma, x, y) \quad \text{root}(y) \in \Upsilon}{\Upsilon; \Gamma; L \vdash \text{dread}(x) : \Gamma(x); \Upsilon}$	$\text{T-FIELD} \frac{\tau_f = \text{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad \tau = \text{Alias}(\tau_f)}{\Upsilon; \Gamma; L \vdash p.f : \tau; \Upsilon}$	
$\text{T-WFIELD-U_GUARDED} \frac{\begin{array}{c} \text{u_guarded}(x) \ c = \text{FieldAccess}(\Upsilon, \Gamma, L, p, f) \\ \Upsilon; \Gamma; L \vdash e : \text{u_root } c; \Upsilon' \end{array} \quad \begin{array}{c} \boxed{y \in \text{MustAlias}(\cdot, e)} \quad \boxed{\text{root}(y) \in \Upsilon'} \quad \boxed{\Upsilon'; \Gamma; L \vdash r \mapsto p} \quad \boxed{r \mid y \in \Upsilon'} \\ \Upsilon; \Gamma; L \vdash p.f := e : \text{borrowed}(x) \ c; \Upsilon' / y \end{array}}{\Upsilon; \Gamma; L \vdash p.f := e : \text{borrowed}(x) \ c; \Upsilon' / y}$		
$\text{T-WFIELD-GUARDLESS} \frac{\text{guardless } c = \text{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad \Upsilon; \Gamma; L \vdash e : \text{guardless } c; \Upsilon'}{\Upsilon; \Gamma; L \vdash p.f := e : \text{guardless } c; \Upsilon'}$		$\text{T-SUB-TYPE} \frac{\Upsilon; \Gamma; L \vdash e : \text{borrowed}(x) \ c; \Upsilon'}{\Upsilon; \Gamma; L \vdash e : \text{guardless } c; \Upsilon'}$
$\text{T-DFIELD} \frac{\text{u_guarded}(x) \ c = \text{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad \text{FinalAlias}(\Gamma, p, f, t) \quad t \notin \Upsilon \quad \Upsilon; \Gamma; L \vdash r \mapsto p \quad \Upsilon' = \Upsilon \cup \{\text{root}(t)\} \cup \{t \mid \forall z. r \mid z \in \Upsilon'\}}{\Upsilon; \Gamma; L \vdash \text{dread}(p.f) : \text{u_root } c; \Upsilon'}$		
$\text{T-NEW} \frac{\boxed{\text{root}(z) \in \Upsilon \vee \text{subtree}(z) \in \Upsilon} \quad \Upsilon_{\text{body}} = \Upsilon \cup \{\text{root}(y)\} \cup \{y \mid z\} \cup \{y \mid t \mid \forall t. t \mid z \in \Upsilon\} \quad \Upsilon_{\text{body}}; \Gamma[x \mapsto \text{u_root } c] \mid \boxed{y \mapsto \text{final guardless } c} : L \vdash e : \tau; \Upsilon'}{\Upsilon; \Gamma; L \vdash \text{let } \text{u_root } x, \text{ final } y = \text{new } c \text{ in } e : \tau; \Upsilon' / y}$		
$\text{T-LET} \frac{\Upsilon; \Gamma; L \vdash e_1 : \tau_1; \Upsilon' \quad \Upsilon'; \Gamma[x \mapsto q \ \tau_1]; L \vdash e_2 : \tau; \Upsilon''}{\Upsilon; \Gamma; L \vdash \text{let } q \ x = e_1 \ \text{in } e_2 : \tau; \Upsilon'' / x}$		$\text{T-SPAWN} \frac{\Gamma(x) = \text{nonfinal } \alpha \ c \quad \emptyset; \{x \mapsto \text{final guardless } c\}; \emptyset \vdash e : \tau; \Upsilon'}{\Upsilon; \Gamma; L \vdash \text{spawn } x \ e : \text{guardless } c; \Upsilon}$
$\text{T-LOCK-FIRST} \frac{\text{FinalAlias}(\Gamma, p, x) \quad \boxed{\Upsilon_t = \text{NewSubtrees}(\emptyset, \Upsilon, x)} \quad \boxed{\neg \exists z. \text{subtree}(z) \in \Upsilon} \quad \Upsilon \cup \Upsilon_t; \Gamma; [x] \vdash e : \tau; \Upsilon' \quad \forall y \forall c. \tau \neq \text{borrowed}(y) \ c}{\Upsilon; \Gamma; \emptyset \vdash \text{lock } p \ e : \tau; \Upsilon' / \{z \mid z \in \Upsilon_t\}}$		
$\text{T-LOCK-N} \frac{\Upsilon; \Gamma; L \vdash p : \tau_1; \Upsilon \quad \tau_1 = \text{borrowed}(y) \ c \quad \text{RaceFreePath}(\Gamma, L, p) \quad \text{FinalAlias}(\Gamma, p, x) \quad \boxed{\text{ValidCap}(\Gamma, L, p, y)} \quad \Upsilon; \Gamma; x :: L \vdash e : \tau; \Upsilon' \quad \boxed{y \in L} \quad \forall z \in L : L = L' @ [z] \vee \exists p'. \text{RaceFreePath}(\Gamma, L, p') \wedge \text{FinalAlias}(\Gamma, p', z) \wedge \Gamma; L \vdash p' : \text{u_guarded}(a) \ c \quad \forall w \forall c. \tau = \text{borrowed}(w) \ c \Rightarrow w \in L}{\Upsilon; \Gamma; L \vdash \text{lock } p \ e : \tau; \Upsilon'}$		

Figure 6. Source typing. MustAlias() is an external must-alias analysis, explained in Section 3.3.

- T-LOCK-FIRST types a thread’s first lock acquisition: type the path being locked, find a final alias for the lock, and type the body in the extended environment. This rule may add a subtree assertion if Υ does not contain a root assertion for the target lock (else no modifications to the tree would be possible in the body). The last hypothesis requires that the resulting type is well-formed with respect to the current lock set (does not borrow from a field of an unlocked object). The set of variables removed from Υ' in the conclusion has cardinality 0 or 1 depending on whether Υ_t added a subtree assertion.
- T-LOCK-N is similar to T-LOCK-FIRST. However, this rule also enforces that locks acquired after the first must be safe. It uses $\text{ValidCap}(\Gamma, L, p, y)$ to check that the borrowed reference being locked is valid, and checks that the thread holds the capability to acquire the target lock through $y \in L$. Finally, it checks an approximation of the “orphaned lock” criteria (Section 3.5) to ensure the thread does not lock the ancestor of an orphaned lock: each lock must be the first lock acquired, or must be accessible at u_guarded type through some race-free path (implying that its parent is still locked, so it is not orphaned).

4. Examples and Extensions

This section explains how lock capabilities can be used to verify our motivating examples. We also discuss extensions to the core system

that may be necessary to capture those examples precisely. The accompanying technical report [13] includes additional examples.

4.1 Tree Rotations

Figure 1 implements the clockwise rotation operation in a splay tree. Splay trees are self-balancing binary search trees with the additional property that recently-accessed elements are faster to look up: a lookup performs a series of rotations to lift the found element to the root of the tree. A fine-grained locking implementation of a splay tree would actually need to hold locks all the way from the root of the tree to the located element. Thus it is a poor candidate for fine-grained synchronization because external pointers to interior nodes are not practically useful. However, we show it here because it has been used to demonstrate the flexibility of other deadlock freedom systems [5], because it is a challenging benchmark for expressiveness, and because it reflects similar issues to those seen in more practical examples, such as other binary trees or reordering the elements in a linked list.

For each lock acquisition after the first, the type system ensures that the target lock is guarded by the capability of an already-held lock. In the innermost critical section, the type system keeps track of the fact that the capability-granting trees rooted at n , x , v_name , and w_name (the latter two being final guardless references) are mutually disjoint after the destructive reads, and can therefore check

that the capability-granting changes implied by storing the unique references back in the heap preserve the relation’s acyclicity.

Figure 1’s code shows how flexible the use of `u_guarded` references for strong updates is: nowhere does the code need to state what the new guard on any lock is. Instead the changes to guards are implicit in the heap changes that induce the new capability grants. It also shows the value of leveraging must-alias information in the type system. This example cannot be expressed in a system that does not support must-aliasing, because there is no way to express the node reordering operations in a way that does not change the meaning of some path expression rooted at `n` before it must be used. Without must-aliasing information it is impossible to track the capability-granting tree structures. This example does not demonstrate flexibility of locking multiple locks with the same guard, but shows that we can verify an example that only two non-standard extensions to the standard static deadlock freedom approach can verify [5, 16]. While there is syntactic overhead from requiring that any reference locked or unique reference moved must alias a final variable, elaboration of a source language to an intermediate language with these properties is straightforward.

4.2 Arrays

Treated as a structure with integer-named fields containing `u_guarded` references, the array elements behave as the children in the simple binary tree example. Locking one array element is always safe, but acquiring multiple elements would require holding the lock on the array itself. Concretely, a final reference `arr` to an array of type `u_guarded Object []` (an array of `u_guarded` references, in a core language extended with arrays) might through dereference (as in `arr[i]`) produce elements of type `borrowed<arr> Object`, which could be locked in any number in any order while a lock is held on the array `arr` itself.

This does trade some potential parallelism for verifiability, because rather than ensuring proper ordering on the array elements this solution makes it safe to not use ordering by protecting the ability to lock elements with the array’s lock; in cases where locking multiple elements of the array is relatively rare dynamically, this would be acceptable. The dominant static deadlock freedom approach offers no solution for locking multiple array elements. The only prior solution we are aware of for statically proving deadlock freedom when locking multiple array elements is a technique that synthesizes additional synchronization to avoid deadlock [21]. For arrays, that technique can synthesize locking equivalent to our solution, though their iterative process can also over-synchronize (see Section 6.3). Another possible approach is to lock cell referents in order of increasing index. Unfortunately, the relative ordering of dynamically-computed array indices is undecidable.

4.3 Method Calls

Extending our core language with method calls is relatively straightforward. For reasons of space, we only sketch this extension. Our mechanism is inspired by Haller and Odersky’s capability-based invalidation mechanism for borrowed references [14]. The key to using borrowed and unique references as arguments to methods is ensuring that no two borrowed arguments are aliases of each other. In our core language, there is the additional matter of the tree assertions a method might require upon entry and provide upon return, and assumptions about held locks — all things typically documented informally in code today.

In the core system without methods or loops presented here, L happens to exactly describe the locks held dynamically. However our type system naturally supports polymorphism over Υ and L . The initial and final disjointness assertions only need to be partial; the implementation may lose information before returning. In particular, extra disjointness assertions in the initial Υ are safe, and

```
class CircularList {
  fixed<this> CircularListNode<this> head;
}
class CircularListNode<ghost CircularList list> {
  fixed<list> CircularListNode<list> prev;
  fixed<list> CircularListNode<list> next;
  u_guarded<this> Object data;
  public fixed<l> CircularListNode<l> find(guardless Object target)
    with <l>, <this>
  {
    if (this.data == target) { this }
    else { let final fnext = this.next in
           lock (this.next) { this.next.findBefore(target, this) }
    }
  }
  private fixed<l> CircularListNode findBefore(
    guardless Object target, fixed<l> CircularListNode<l> start)
    with <l>, <this>
  {
    if (this == start) { null }
    else {
      if (this.data == target) { this }
      else { let final fnext = this.next in
             lock (this.next) { this.next.findBefore(target, start) }
      }
    }
  }
}
```

Figure 7. A circular list, using fixed-guard references and external capabilities.

a form of frame rule for Υ can be proven. For methods requiring certain locks to be held on entry, the method is still safe if additional locks (additional capabilities) are held, with the exception that methods acquiring an arbitrary guardless reference would still require no locks to be held at the call site.

4.4 External Capabilities for Circular Lists

Another example where the flexible acquisition order plays a central role is the circular lists commonly seen in operating system kernels as the process or thread lists. The processes themselves form a circular doubly-linked list. The locking discipline is as follows: locking one node of the list is allowed, while multiple nodes may be locked only if a lock over the whole list is held. There is no consistent acyclic order on the list nodes, short of resorting to memory addresses for sorting. No prior technique for static deadlock freedom verification can verify this example.

Capturing this sort of discipline in our core language requires a small extension for objects’ field types to refer to external capabilities. We can use the same class parameterization as in RCC/JAVA [10] to refer to external locks (another use of final variables for lightweight singleton types), simply using the additional lock names in scope for field types with other locks as guards, rather than field types whose access is protected by another lock. We also require *fixed references*, which are the sort of reference one would expect in a system without lock reordering: non-unique mutable reference types extended with capability information, not subject to strong updates. This allows the list nodes to be guarded by the main list lock without a direct field reference, and allows that guard information to be shared among multiple references to each (doubly-linked) list node.

An example is shown in Figure 7. It shows an in-order traversal of the circular list to locate some element. Because the exposed `find` method requires the capability $\langle l \rangle$, only one thread at a time may execute this code on a given circular list. But other threads may simultaneously access individual list nodes by locking through guardless references to list nodes.

This extension would of course require small changes to field typing, and a way to convert unique references into permanently-fixed references. And because these fixed references define permanent capability-granting relationships unrelated to heap structure, several invariants would need slight adjustments.

4.5 Lock Capabilities with Lock Levels

It is possible to combine lock capabilities with the dominant approach for static deadlock freedom to afford the flexibility of each where necessary. Previous work on statically ensuring deadlock freedom has focused on *lock levels* [5, 9, 16]: a static partitioning of the heap accompanied by a partial ordering on those partitions. A static checker verifies that while a thread holds a lock in a certain level then any additional locks acquired must reside in a level below the held lock. If thread A is blocked waiting for a lock l held by thread B, any locks A already holds are in partitions ordered *before* the partition of l . Since B holds lock l and it may only acquire locks in partitions ordered *after* l 's partition, it cannot block on a lock held by thread A (or another thread transitively blocked on a lock held by A).

This approach suffices for programs using coarse-grained locking such as that between multiple subsystems of a program and, with some extensions, for certain narrow classes of programs using fine-grained locking. But in general lock levels have poor support for most fine-grained locking techniques, for programs that change lock ordering dynamically, and for programs that acquire multiple locks that are related but have no sensible ordering among them. By contrast, lock capabilities are well suited to reasoning about local lock orderings within a set of closely-related locks.

A lock capability system can be run within each partition of a lock level system. Thus a thread may acquire a target lock when it holds no locks; when it holds locks only in levels ordered before the level of the target lock; or when it holds a lock that grants a capability to acquire the target lock, and it holds no locks in levels ordered after that of the granting lock. This allows, for example, use of two fine-grained data structures in different levels. We have not proven safety for this embedding, but expect no subtleties.

4.6 Unstructured Locking

As mentioned in Section 3.5, extending lock capabilities to support unstructured locking primitives (i.e., explicit lock and unlock statements) would require a few changes. First and foremost, the static lock set would need to be flow-sensitive. Second, to take advantage of the flexibility offered by unstructured locking, the criteria for orphaned locks would need to be refined to only prevent acquiring locks that may (transitively) grant the capability to acquire some lock the thread already holds (which is the actual unsafe behavior). In Figure 4, this would mean permitting the thread to use b 's capability to acquire locks after releasing the lock on c , but not permit the use of a 's, while the system presented here would prevent both until the lock on c is released.

Extending to unstructured locking would permit such idioms as hand-over-hand locking through data structures, or even parallel instances of hand-over-hand locking, for example to acquire locks guarded by each of multiple processes in the circular list example. With unstructured locking, the capability-granting structure of the circular list example suggests a verifiably deadlock-free and fairly parallel solution to the well-known Dining Philosophers Problem, detailed in a technical report [13].

5. Soundness

We have proven that our type system ensures deadlock freedom. The full proof is available in the accompanying technical report [13]. This section sketches the proof.

The argument relies on two proofs: type preservation and a separate deadlock-freedom preservation proof that accounts for changes in the capability-granting relation. We define deadlock formally as a cycle of threads each waiting for the next to release a lock. Proving that the absence of such cycles is preserved is equivalent to proving progress up to null dereference. It is possible

$$\begin{array}{c}
 \Sigma : \text{Location} \mapsto \text{Class} \quad \phi : \text{Value} \mapsto \text{Variable} \\
 \text{T-ANY-LOCATION} \\
 \frac{\Sigma; \phi; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon \text{ cont.}}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \text{guardless } c; \Upsilon} \\
 \text{T-UNIQUE-NULL} \\
 \frac{\text{FinalAlias}(\Gamma, \text{null}, y) \quad \text{root}(y) \in \Upsilon}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{null} : \text{u_root } c; \Upsilon} \\
 \text{T-UNIQUE-LOC} \\
 \frac{\Sigma(l) = c \quad \text{FinalAlias}(\Gamma, l, y) \quad \text{root}(y) \in \Upsilon}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash l^* : \text{u_root } c; \Upsilon} \\
 \text{T-BORROWED-VALUE} \\
 \frac{\phi(v) = x \quad p \in \text{MustAlias}(\cdot, v) \quad \Gamma; L \vdash p : \text{u_guarded}(x) \ c}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \text{borrowed}(x) \ c; \Upsilon} \\
 \text{T-WITHLOCK} \\
 \frac{\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \tau_1; \Upsilon_1 \quad \text{FinalAlias}(\Gamma, l, x) \quad \Upsilon_1 = \text{NewSubtrees}(L, \Upsilon, x) \quad \forall z. \text{subtree}(z) \in \Upsilon_1 \Rightarrow \text{subtree}(z) \in \Upsilon_1}{\Sigma; \phi; \Upsilon_1; \Gamma; x :: L \vdash e : \tau_2; \Upsilon_2 \quad \tau_2 = \text{borrowed}(y) \ c \Rightarrow y \in L} \\
 \Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{withlock } l \ e : \tau_2; \Upsilon_2 \ / \{z | z \in \Upsilon_1\}
 \end{array}$$

Figure 8. Typing for runtime expression forms.

for a thread in our system to become permanently “stuck” because it tries to dereference null, or because it blocks waiting for a thread that diverges while holding a lock. Our system is not designed to prevent such errors. It ensures that, modulo null pointers, there is always at least one thread that is not blocked and there are no cycles of threads blocked on each other.

Type preservation is tedious, but mostly straightforward to prove given the run-time type rules and appropriate invariants. The typing rules for run-time expressions extend each rule with a heap typing Σ giving the class for each heap location, and a group typing ϕ specifying the lock group for each location. Σ is global, while there is a separate ϕ for each thread, giving the lock group for those locations whose unique reference the thread controls. The domains of the ϕ s are disjoint, isolating the lock group information necessary for strong updates. Additional invariants include various well-formedness constraints on type contexts, that the capability-granting relation is a forest, that no two threads with root assumptions also have subtree assertions for locks capability-reachable from each other’s roots (used to prove that after adding a capability-granting edge, other threads’ Υ contexts are still valid), and that ϕ for each thread contains all possible names (final variables) for the lock group of each lock whose unique reference the thread controls. These invariants are all part of the typing for runtime states, available in the technical report [13].

The additional typing rules for runtime-only expression are shown in Figure 8. For brevity, we only detail T-WITHLOCK. Most of the rule is similar to the static locking rules, typing the lock, typing the body expression in an extended lock set, and ensuring the resulting type does not borrow from the released lock. The main difference is in the treatment of subtrees (the shaded hypotheses). These ensure that any necessary subtree assertion is already present in the input Υ , necessary for type preservation.

Proof of deadlock freedom relies on what is essentially another preservation proof over an extended semantics. Intuitively, in a system without changes in the capability-granting relation, the deadlock freedom argument is straightforward: because the capability-granting relation is acyclic, the relation of which threads are blocked on which others must also be acyclic because there must always be a thread at the “bottom” of any capability tree which can continue to execute without blocking. Crucial to that simplicity is that any blocked thread would have a path in the graph represented by the capability-granting relation from its first lock acquired to the blocking lock, since the thread must have followed

some contiguous path of capability granting edges (by locking) to gain the capability to acquire the blocking lock, and we are (temporarily) assuming no capability-granting changes.

With changes to the capability-granting relation, the argument is more complex. After acquiring several locks, a thread can split a capability tree. Thus the argument for acyclic locking order among threads is less direct. An important observation is that at the time each new lock is acquired, there is a capability-granting edge from some held lock to the new lock. A dynamic log of each thread’s capability uses should be able to show that in a directed graph of those capability uses, no dependency path exists from one thread’s locks, to another thread’s locks, and back to the first thread’s locks.

We extend program state with such a log, a *capability-use graph* representing the use of capabilities by each thread, and prove that the absence of these problematic paths in the graph is preserved. Such a path must exist in the capability-use graph for a deadlocked program state. Graph vertices are locks that are held ($l \in Ls$) or would be acquired by a thread’s next reduction step (l for a thread whose expression is some $E[\text{lock } l \ e]$). An edge $a \xrightarrow{i} b$ is present if the lock a granted the capability to acquire b at the time thread i acquired b or blocked trying. Intuitively each path through the graph represents a dynamically possible dependency chain of threads blocking on locks, for the program being executed. In this graph, there is a path from the first lock acquired by a thread to its most recent acquisition, even with changes to the capability-granting relation. A deadlock manifests in this graph as a path between locks held by the same thread that traverses edges from at least one other thread: either a cycle among threads or a path that leaves the edges of a thread and returns without a cycle. The cycle case is prevented by acyclicity of the capability-granting relation, and the straight line path is prevented by the “orphaned lock” premise of T-LOCK-N. Preserving the absence of such a path in the capability-use graph preserves deadlock freedom.

We believe this proof approach can be extended to support unstructured locking, reader-writer locks,² and embedding within lock levels (as in Section 4.5), with only minor changes. The only assumptions the proof approach makes about what the type system enforces are that the capability-granting relation is acyclic, and that the type system prevents the use of capabilities that may reach orphaned locks.

6. Related Work

Our system builds on work in several areas, including prior approaches to static deadlock detection and work on linear and unique types. For space reasons, we focus on static deadlock freedom and strong updates. For an extended discussion of related work on these topics as well as on deadlock freedom for message passing systems [15, 17], hybrid static/dynamic deadlock freedom [4, 11, 12], static data race freedom [10], capability systems [20], and static aliasing analyses [7], see the accompanying technical report [13].

6.1 Lock Levels

As mentioned in Section 4.5, lock levels [5, 9, 16] is the most prevalent static deadlock freedom approach. Lock levels are adequate for coarse-grained lock ordering, such as between layered subsystems of a program. However, they are ill-suited for dynamic data structures using fine-grained locking. The lock levels approach typically places two serious limitations on code:

²We have not proven the extension sound, but believe splitting capabilities for read and write acquisitions is straightforward: multiple threads can safely possess the same read-acquire-capability because read-locks are not exclusive. The first write-lock must either occur when no locks are held, or be a lock held for reading. Standard locks should be treated as write locks.

The **total ordering restriction**, that there must exist a total ordering among any locks held concurrently by a single thread, is problematic in any structure where there is no natural ordering on locks acquired, such as the circular list example; or where the natural order may be difficult to decide, such as for array-order locking. Some desired locking behavior is difficult to express with lock levels; locking multiple children of a tree node can be expressed with a parameterized lock level system that fixes for a node in level n , `node.left` in level $n + 1$ and `node.right` in level $n + 2$. But even that solution is brittle; `node.left.left` is unordered with respect to `node.right`. This restriction is present in all systems using lock levels.

Systems with the total ordering restriction do not have orphaned lock issues; problematic cases are precluded by requiring that a newly acquired lock be ordered after *all* locks held.

The **fixed ordering restrictions**, which prevent changing the safe acquisition orders of locks, cause problems in algorithms that change orders dynamically, as is the case for many tree structures. It is conceptually the result of two sub-restrictions. There is no known way to lift the **partition ordering restriction** in lock level systems: the inability to reorder partitions. Lifting the **partition membership restriction**, the inability to move locks among partitions, requires control of aliasing. To the best of our knowledge, the partition membership restriction is present in all but two pieces of lock levels work: SAFEJAVA [5], which uses ad hoc extensions and an unspecified flow-sensitive analysis for tracking heap shape; and CHALICE [16], which uses fractional permissions [6] to control access to a ghost variable that defines an object’s lock level.

Leino et al. describe CHALICE, a system using a novel variation on lock levels to avoid deadlock [16]. It avoids some reordering problems in systems with fixed spacing of lock levels by using a dense lattice of unnamed lock levels (any two ordered lock levels have levels between them), and using relative clauses for threads with full permission for a lock’s level ghost variable to reorder a lock relative to other locks. CHALICE also uses fractional permissions [6] to control sharing and modification of not only data, but also the ghost field defining an object’s level. It still enforces the total ordering restriction, and because lock acquisition requires partial access to the level field CHALICE cannot support objects that are both reorderable (which requires full permission) and acquirable by arbitrary threads (requires sharing small permissions to each thread), such as the children from the four thread example in Section 1. CHALICE can verify programs that, for example, reverse the order of a linked list’s nodes while still permitting acquisition in list order. CHALICE allows some controlled sharing of relative ordering information using fractional permissions; lock capabilities could be adapted for this sharing, either by treating a lock’s guard as a ghost field with fractional permissions [6] or using a counting permission type system [3] to track propagation.

6.2 Other Deadlock Freedom Approaches

Attiya et al. [1] describe core results about the range of locking protocols for data structures whose representations are completely inaccessible outside a module — and are therefore strongly encapsulated — that can be verified as deadlock-free and atomic using only sequential reasoning. Their results include the DYNAMIC TREE LOCKING protocol, which is similar to our locking protocol but different in several important ways. First, their technique is applicable only to strongly encapsulated data structure implementations; implementations can acquire only a single lock visible outside the module. Second, they assume total uniqueness of references to module-private locks: no aliasing at all. Third, they consider heap structure to be the only direction of lock ordering. Any of these three is enough to prevent verifying the circular list example in the OS context, where process locks must be aliased and

exposed to all subsystems, and the locking order is not controlled by heap shape. The soundness of their approach depends on the assumption that module-private locks are acquired only when *totally unreachable* by other threads, rather than allowing some aliasing without exposing a capability grant that may change.

While Attiya et al. briefly describe experience with a verifier, their protocol is described axiomatically in terms of dynamic thread behavior, while we provide a concrete type system enforcing correct behavior. Additionally, because their work also ensures that module operations are atomic, they do not permit reacquisition of locks released during the same transaction, which avoids orphaned lock issues, but also prevents treatment of condition variables. Our type system could be paired with an atomicity analysis, and our treatment of orphaned locks naturally supports reasoning about release and reacquisition of one lock while others are held, which is needed for any extensions to handle condition variables.

Attiya et al.'s work grew out of work from the database community [8] on locking protocols for locking in databases of directed graphs. This work focused on serializability in a database under the assumption that locking protocols would be dynamically enforced by a central concurrency management system.

Wang et al. [22] use a translation of programs to and from petri nets to synthesize additional locking that avoids deadlock. Their synthesized synchronization often resembles locking arrangements checkable by lock capabilities. For example, they synthesize solutions equivalent to our array locking solution [21]. Because their process is iterative, multiple layers of locks may be synthesized, leading to results that could be unpredictable, though in their experiments overhead was low. Lock capabilities require explicit decision to use similar constructs, and the fact that the similar locking they synthesize tends to be cheap suggests use of lock capabilities will not impose undue runtime overhead. Their approach is also highly sensitive to aliasing; in the worst case they generate locks that guard all locks of a certain type, which would be prohibitive for code using multiple recursive data structures.

6.3 Other Type Systems

Gerakios et al. [11, 12] describe a type and effect system that enforces correct use of a locking primitive that, before acquiring a target lock, checks that all future locks that will be acquired before the target is released are also available. When checking sub-effecting for functions, their approach is highly conservative, and likely to cause problems with separate compilation by requiring strong over-approximations of locks-to-be-acquired at each program point. They also present benchmark numbers, but do not discuss using their implementation (which uses a locking primitive at least linear in the number of future lock acquisitions before release) on benchmarks that acquire more than two locks, such as tree traversals. For more complete discussion, see our technical report [13].

Bierhoff and Aldrich describe a set of *access permissions* for use in combining tpestate (which require a form of strong updates) with aliasing [2], where each access permission implies different permissions for changing an object's tpestate. Our *u_guarded* and *guardless* references roughly correspond to their full and pure permissions. The fixed qualifier we propose in Section 4.4 roughly corresponds to their share permissions. Bierhoff and Aldrich's system is somewhat more general than the guardless uniqueness we describe: because they use fractional permissions to track the degree of splitting, in cases where aliasing remains controlled, it is possible to convert in both directions between full/pure (*u_guarded*/guardless) references and share/pure (fixed/guardless) references by combining all references back into a single unique reference, then splitting again into the other reference types. Adapting their rich conversion logic could help us verify DAG-shaped capability-granting relations.

7. Conclusion

We have described a capability-based approach to deadlock freedom that is more flexible than existing locking protocols, and a type system for a core language that enforces this locking protocol. We support dynamic changes in acquisition order, and in some cases absence of a relative ordering between locks that may be held concurrently. We have also described subtle soundness issues the system avoids, and contrasted our work with related work on lock levels and deadlock freedom in strongly encapsulated modules. Our system forms the basis for a deadlock freedom verification approach that is suitable for many complex locking protocols.

Acknowledgments

This work was supported by NSF Grant CNS-0855252. We also thank members of the UW PLSE group and the anonymous referees for helpful comments on improving this work.

References

- [1] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential Verification of Serializability. In *POPL*, 2010.
- [2] K. Bierhoff and J. Aldrich. Modular Tpestate Checking of Aliased Objects. In *OOPSLA*, 2007.
- [3] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *POPL*, 2005.
- [4] G. Boudol. A Deadlock-Free Semantics for Shared Memory Concurrency. In *ICTAC*, 2009.
- [5] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.
- [6] J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, 2003.
- [7] M. Bravenboer and Y. Smaragdakis. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA*, 2009.
- [8] V. K. Chaudhri and V. Hadzilacos. Safe Locking Policies for Dynamic Databases. In *PODS*, 1995.
- [9] C. Flanagan and M. Abadi. Types for Safe Locking. In *ESOP*, 1999.
- [10] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.
- [11] P. Gerakios, N. Papaspyrou, and K. Sagonas. A Type System for Unstructured Locking that Guarantees Deadlock Freedom without Imposing a Lock Ordering. In *PLACES*, 2010.
- [12] P. Gerakios, N. Papaspyrou, and K. Sagonas. A Type and Effect System for Deadlock Avoidance in Low-level Languages. In *TLDI*, 2011.
- [13] C. S. Gordon, M. D. Ernst, and D. Grossman. Static Lock Capabilities for Deadlock Freedom. Technical Report UW-CSE-11-10-01, Computer Science and Engineering, University of Washington, Seattle, WA, USA, 2011.
- [14] P. Haller and M. Odersky. Capabilities for Uniqueness and Borrowing. In *ECOOP*, 2010.
- [15] N. Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR*, 2006.
- [16] K. R. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, 2009.
- [17] K. R. Leino, P. Müller, and J. Smans. Deadlock-free Channels and Locks. In *ESOP*, 2010.
- [18] F. Smith, D. Walker, and J. G. Morrisett. Alias Types. In *ESOP*, 2000.
- [19] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *TIC*, 2000.
- [20] D. Walker, K. Crary, and G. Morrisett. Typed Memory Management via Static Capabilities. *ACM TOPLAS*, 22, 2000.
- [21] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*, 2008.
- [22] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The Theory of Deadlock Avoidance via Discrete Control. In *POPL*, 2009.