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 is built around 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, and in data-structures where heap shape affects safe locking orders, we can use the heap structure to 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.

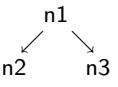
1. Introduction

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 soundly combined to yield a yet more expressive system.

1.1 Fine-grained Locking and Deadlock Freedom

A long line of research develops techniques that deal well with avoiding 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 [6, 19] (precluding thread 3 or thread 4), or assume strong encapsulation for recursive structures [2] (precluding the interior pointers n2 and n3 of threads 1 and 2). Verifying deadlock freedom for fine-grained locking becomes even more difficult when considering 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 [2, 6, 19] can verify deadlock freedom in binary trees whose structure changes over time, such as splay trees (shown in Figure 1 and discussed in Section 4.2).

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 4.3.

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 4.7.

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.

* Extended Version; Technical Report UW-CSE-11-10-01, Computer Science and Engineering, University of Washington

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

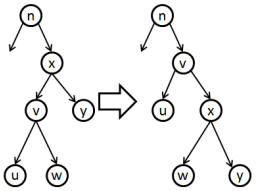


Figure 1. Splay tree rotation in the core language. Assume `n` refers to the node from which to rotate. `dread` is a destructive read, and variables may be bound as `final` (i.e. immutable) or `nonfinal`. The extra final variables provide names for the type system to track.

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 the circular list above: suppose that each list node is guarded by the same per-list capability; then threads that must hold multiple process locks simultaneously are serialized via the per-list lock, but parallelism with threads that require only one node'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 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. Using traditional unique references prevents sharing a structure across threads, so we introduce the concept of partially-unique references. A *complete* reference to an object is the only (unique) reference with the guard in its type. Other references — *partial* references — may alias the complete 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 complete references naturally expresses the desired changes in the capability-granting relation. For examples like the splay tree in Figure 1, straightforward mutations with the complete 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 4.7).

[1] Reordering 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 restrictions necessary to handle such cases soundly are described in Section 3.5.

Programs	P	::=	$\overline{class}\ e$
	$class$	::=	class c $\{\overline{field}\}$
	$field$	::=	$\tau\ f$
Paths	p	::=	$x \mid p.f$
Expressions	e	::=	if $e\ e\ e \mid p := e \mid$ lock $p\ e$
		$\mid$	dread$(p) \mid$ spawn $x\ e \mid$ null $\mid p$
		$\mid$	let $q\ x = e$ in e
		$\mid$	let guardless x, final $y =$ new c in e
Final Qualifiers	q	::=	nonfinal $\mid$ final
Types	τ	::=	$\alpha\ c$
Base Types	α	::=	$u \mid$ partial $\mid$ borrowed$\langle x \rangle$
Unique Types	u	::=	guardless $\mid$ complete$\langle x \rangle$

$x, y, z \in$ Variables $\quad f \in$ Fields $\quad c \in$ Classes

Figure 2. Core language syntax.

In this paper we implement the lock capabilities verification approach as a static capability type system [28]: 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 [17] 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 [13] to name locks and their associated static capabilities consistently.

1.3 Contributions

- We present an alternate technique for static deadlock freedom verification — lock capabilities — that fits well with fine-grained locking, allows flexible lock acquisition orders, and supports changing acquisition orders at runtime.
- We present a type system to verify deadlock freedom statically using lock capabilities (Sections 2 and 3).
- We introduce partially-unique references, the first extension of unique references we are aware of that permits some strong updates at the level of individual objects, in the presence of unlimited aliasing (Section 3.2).
- We illustrate how lock capabilities enable static verification of deadlock freedom 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 4).
- We prove soundness for lock capabilities in a type-and-effect system (Section 5).
- We compare lock capabilities to prior work on static deadlock prevention, and discuss how lock capabilities address limitations of that work (Section 6).

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. Figure 2 presents the syntax. Aside from the types, the language is mostly straightforward, containing conditionals, mutable local 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, local variable binding, and allocation (with the result bound

to two locals — see below). We focus on structured locking in this paper, rather than unstructured (i.e. explicit lock and unlock statements). This is not a fundamental restriction, and extension to unstructured locking is discussed in Section 4.9.

The types include two types of unique references: $\mathrm{complete}\langle x\rangle$ references are unique references that associate their referent with a capability and are stored only in fields, and guardless references are unique references whose referents are not associated with any capability and are stored only in local variables. $\mathrm{dread}(p)$ 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 complete reference to a guardless reference by destructively reading a field, and to convert the other direction by storing a guardless reference into a complete field. These operations break and create associations, respectively, between objects and guarding capabilities. There are also borrowed references (temporary aliases of complete references), $\mathrm{borrowed}\langle x\rangle$, which may only be stored in local variables; and regular references with no duplication restriction, partial references, which may be stored in fields or local 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 (immutable) variable bound to a partial reference. The final reference is necessary to name the new object in the type system as soon as it is allocated; this is not uncommon in type systems with simple singleton types [13, 25, 27]. 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 complete or guardless. 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 next expression e of thread tid in the provided heap H, environment V, and lock set Ls.

We assume an α-renamed program, so no two `let` expressions bind the same variable.

Most of the rules are standard, with a couple minor exceptions:

- 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 local variables rather than performing binding by substitution. This simplifies the semantics for destructive reads. We use the notation $\mathsf{Dup}(v)$ to preserve uniqueness of tagged value; specifically, it ensures there is never more than one $l^\bullet$ for any l in the heap's domain ($\mathsf{Dup}(l^\bullet) = l$).
- E-*LOCK: The semantics support recursive lock acquisition. When an object's implicit lock is not held, the owner field of its heap value is None. When the owner field is $\mathrm{Some}(tid)$, the thread holding the lock has identifier tid (E-LOCK). A thread can acquire a lock it already owns (E-RECLOCK). 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.

Methods could be added to the dynamic semantics in a straightforward manner (see Section 4.4), but have been omitted to simplify presentation.

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.

Table 1 lists the major invariants preserved by the type system. Each invariant will be explained in detail as it arises.

The main typing judgement is $\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$. Here Υ 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, Υ is consulted to verify that a modification will not introduce a cycle. Because the relation can change, Υ' is the new capability-granting relation after executing e. Treatment of Υ and Υ' is detailed in Section 3.6. Γ maps local variables to their types, with a qualifier describing variables as final (i.e., immutable) 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 Υ hold, Γ provides accurate types for local 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 Υ' 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 in detail the "orphaned lock" problem introduced by early lock releases and by changing the capability-granting relation. Section 3.6 explains how the type system ensures that the capability-granting relation remains acyclic, and Section 3.7 introduces the typing rules.

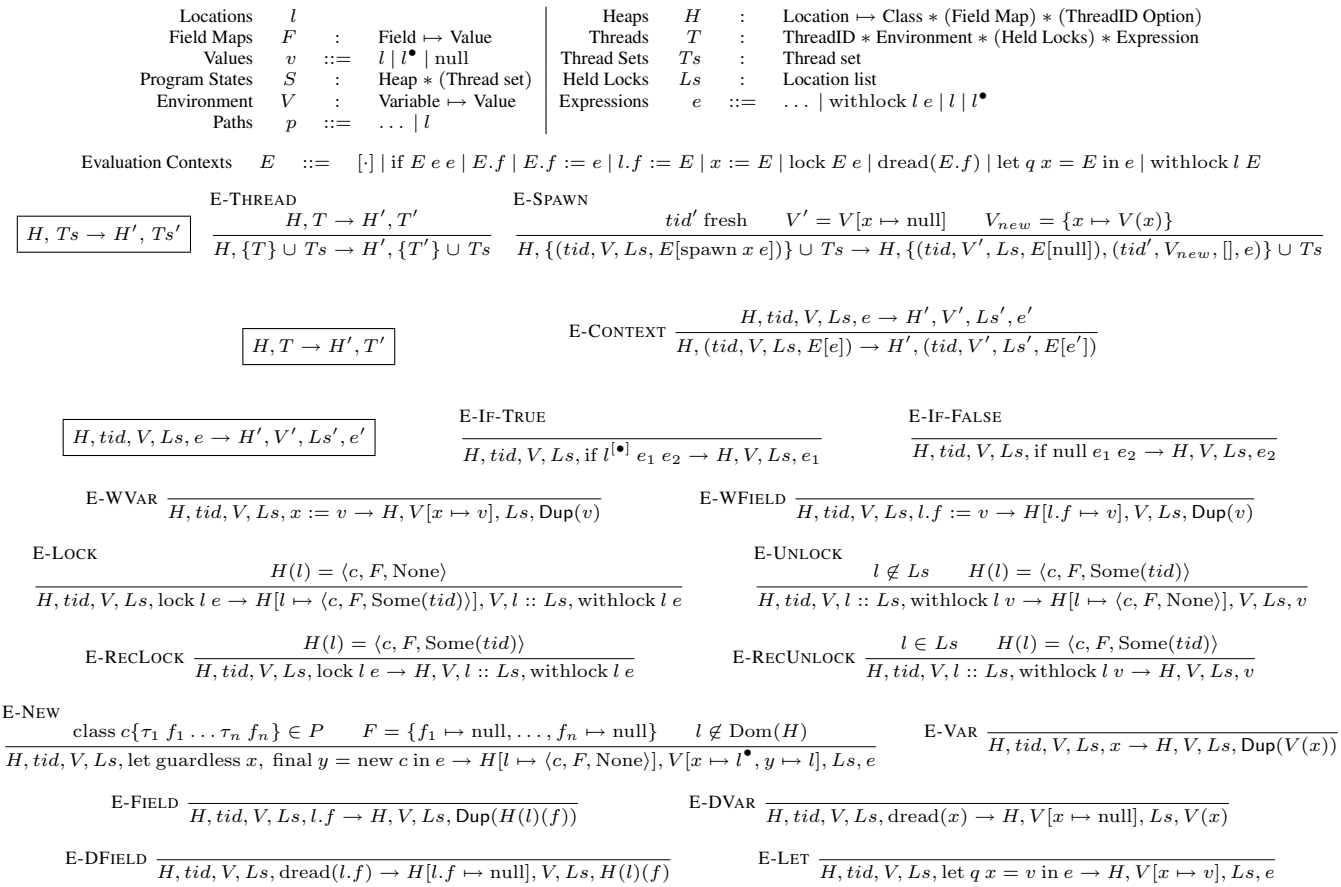


Figure 3. Operational Semantics

Name	Property	Section
Lockset Approximation	Each final variable x in the static lock set L corresponds to a lock l in the dynamic lock set Ls	3.1
Single Guards	Each object is guarded by at most one capability	3.2
Capability Forest	The capability-granting heap edges form a forest; in other words, the heap, restricted to those fields the program declares as complete, is a forest	3.3, 3.6
Lock Valid Guards	If the target lock of a `lock` statement is a borrowed reference, that reference will be valid (the capability information on the borrowed reference matches the corresponding complete reference) when execution reduces that expression	3.2
Capability Grant Partitioning	For each dynamic location, at most one thread's context associates it with a guarding lock (this ensures that other threads still type check after a strong update to an object's guard)	3.4

Table 1. Listing of major invariants in the type system.

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 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 literature [13, 25, 27].

L is a static approximation 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. 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.

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 mem-

bership 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 type of the target lock is $\mathrm{borrowed}\langle y \rangle\ c$ in the 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, $\mathrm{complete}\langle x \rangle\ c$ 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 $\langle \mathrm{this} \rangle$ because this is the only capability name in scope in that context; Section 4.7 describes an extension for using other objects' capabilities. $\mathrm{guardless}\ c$ 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 complete references (as in $\mathrm{complete}\langle x \rangle\ c$) to perform strong updates to the guard portion ($\langle x \rangle$) of an object's type: destructively reading a complete field removes the referent from its previous parent's lock group, and storing a guardless reference into a complete field moves the referent into its new parent's lock group. Object fields may not be $\mathrm{guardless}$ references, and local variables may not be complete references. For example, assuming `x` is a final variable with complete field `f`:

```
let nonfinal y = dread(x.f) in
// y is guardless, x.f holds null
x.f := dread(y)
// x.f contents again complete<x>, y holds null
```

Complete references are necessary only for controlling the capability-granting relation. Most references are normal, and do not need to carry the referent's guard. We call these partial references (with type $\mathrm{partial}\ c$). Note that we allow partial references to objects for which there is also one unique reference. This is sound because with a partial reference only reads, writes, and acquiring a first lock are permitted. In a typical program, most references would be partial or some other non-unique qualifier, such as thread-local.

"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. Supporting 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 same inconvenient idiom as many type systems with linear types: linear/unique items that need to be used but eventually reside in their original location end up being explicitly threaded through the computation rather than being passed normally.

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

A unique reference's lock group information is always valid, 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 complete 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 $\mathrm{borrowed}\langle x \rangle\ c$ 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 complete reference, we use the metafunction $\mathsf{ValidCap}(\Gamma, L, p, x)$ (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.

Concretely:

```
// assume x is final, with complete field f
let nonfinal y = x.f in // y:borrowed<x>
// y's guard is valid,
// only because it is aliased to x.f
let nonfinal a = dread(x.f) in // a:guardless
// Guard information on y is now invalid,
// so the next line would be a type error
// let nonfinal _ = y in
let nonfinal b = a in
// b:partial, since a:guardless
...
```

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 [25, 27]. However, 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 $\mathsf{MustAlias}(\cdot, p)$ 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 preserves and propagates 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: $\mathsf{MustAlias}(\cdot, p) \equiv \mathsf{MustAlias}(\theta, p_\theta)$. The must-alias analysis is queried throughout the type judgements, both explicitly, and implicitly through the use of some macros defined 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 through the use of the macro $\mathsf{RaceFreePath}(\Gamma, L, p)$ throughout the typing judgements.

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 complete references, we also maintain the invariant that for each dynamic location, at most one thread's typing context associates it with a lock group.

Note that while we enforce data race freedom, it is not strictly necessary. Allowing data races on partial reference fields does not risk deadlock; a thread may only lock a partial 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 complete references can be made sound if any unsynchronized read produces a partial reference (synchronized reads may still return borrowed references). What truly must be race free is the use of the guard information encoded in complete fields. Our enforcement of complete 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.

```
... // 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!
    }
  }
}
```

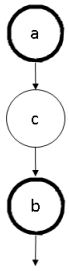


Figure 4. Java code demonstrating an "orphaned lock" (lock b) in the context of a singly-linked list. The diagram on the right shows the heap (and capability-granting) structure after the first lock release. The nodes with dark borders are locked at that point by the thread running the code above. A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

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 complete reference to a held lock other than the first acquired (in Figure 4, b is orphaned because the thread lacks access to the complete reference for c). 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 performed by operations

on complete references. Table 2 summarizes these tree operations in terms of operations from an initial graph G to a new graph G', and the typing rules that model these operations in Υ.

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 complete reference edges. Extensions with capability-grants unrelated to heap structure are considered in Sections 4.7 and 4.8. The type system treats destructive reads of complete references as edge removals. Similarly, storing a complete reference is equivalent to adding an edge.

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 $x||y \in \{y||x\}$.
- $\mathrm{root}(y)$ 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 complete reference, or allocating a new object. No two roots in Υ may alias each other.
- $\mathrm{subtree}(x)$ is a weaker substitute for $\mathrm{root}(x)$ when it is unknown whether there is a complete reference to x in the heap, as is often the case when a thread acquires its first lock via a partial reference. It acts as an anchor to keep Υ well-formed. For every $x||y \in \Upsilon$, either $\mathrm{subtree}(x) \in \Upsilon$ or $\mathrm{root}(x) \in \Upsilon$, 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. When the first lock acquired is reached via a partial reference, $\mathsf{NewSubtrees}(L, \Upsilon, x)$ (Figure 5) adds a subtree assertion if the lock is not known to be aliased to some root, so mutations while the lock is held may produce disjointness assertions relative to the first lock.

The typing rules use a judgement $\Upsilon; \Gamma; L \vdash x \rightarrowtail p$ 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. Allocation adds disjointness assumptions relative to only one subtree or root rather than all local and global roots. This is because there may be a subtree and a root in Υ whose variables alias each other, and adding assertions that a new allocation is disjoint from both variables could allow a cycle to be created:

```
// Assume Υ contains root(a), subtree(b)
// Assume a and b alias, but the must-alias analysis
// does not know it.
// a and b are final variables (they appear in Υ),
// a_alias is a nonfinal alias of a
let unique ou, final o = new Object in
  // Say we add both o||a and o||b to Υ
  o.f = dread(a_alias); // Removes all uses of a from Υ
  // Υ now contains o||b even though o.f==b
  b.f = dread(ou); // A cycle!
```

Having destructive reads of complete references produce guardless references avoids some additional reasoning about trees. The alternative for the type-level effects of destructive reads would be to put the referent lock in an arbitrary lock group, *subject to checks that the new group would not introduce a cycle in the capability-granting relation*. These checks would be the same currently used to ensure that the strong update implied by storing a unique reference to a complete-typed field is safe. Our choice to have destructive reads produce guardless references is equal in expressiveness to a system that picks an arbitrary safe group, but our choice produces a simpler type system.

$$\mathsf{Alias}(\tau) = \begin{cases} \mathrm{borrowed}\langle x\rangle\ c & \text{if } \tau = \mathrm{complete}\langle x\rangle\ c \\ \mathrm{partial}\ c & \text{if } \tau = \mathrm{guardless}\ c \\ \tau & \text{otherwise} \end{cases}$$

$$\mathsf{FinalAlias}(\Gamma, p, x) \equiv x \in \mathsf{MustAlias}(\cdot, p) \wedge \Gamma(x) = \mathrm{final}\ \tau_x$$

$$\mathsf{LockedFinalAlias}(\Gamma, L, p, x) \equiv \mathsf{FinalAlias}(\Gamma, p, x) \wedge x \in L$$

$$\mathsf{RaceFreePath}(\Gamma, L, p) = \begin{cases} \text{true} & \text{if } p = l \\ \text{true} & \text{if } p = x \\ \text{true} & \text{if } \left(\begin{array}{c} p = p'.f \wedge \\ \mathsf{RaceFreePath}(\Gamma, L, p') \wedge \\ \exists x.\mathsf{LockedFinalAlias}(\Gamma, L, p', x) \end{array}\right) \\ \text{false} & \text{otherwise} \end{cases}$$

$$\mathsf{FieldAccess}(\Upsilon, \Gamma, L, p, f) = \tau \text{ if } \Upsilon; \Gamma; L \vdash p : \tau_1; \Upsilon \wedge \mathsf{RaceFreePath}(\Gamma, L, p) \wedge \mathsf{LockedFinalAlias}(\Gamma, L, p, x) \wedge P \vdash \tau_2\ f \in \mathrm{Fields}(\mathrm{Class}(\tau_1)) \wedge \tau = \tau_2[x/\mathrm{this}]$$

$$\mathsf{ValidCap}(\Gamma, L, p, x) = \exists p'.\left(\begin{array}{c} p' \in \mathsf{MustAlias}(\cdot, p) \wedge \\ \Gamma; L \vdash p' : \mathrm{complete}\langle x\rangle\ c \end{array}\right)$$

$$\mathsf{NewSubtrees}(L, \Upsilon, x) = \text{if } (L \neq [] \vee (\exists t.\mathrm{root}(t) \in \Upsilon \wedge t \in \mathsf{MustAlias}(\cdot, x))) \text{ then } \emptyset \text{ else } \{\mathrm{subtree}(x)\}$$

$\boxed{\Upsilon; \Gamma; L \vdash x \rightarrowtail p}$

$$\textsc{Root-Root}\ \frac{\mathrm{root}(x) \in \Upsilon}{\Upsilon; \Gamma; L \vdash x \rightarrowtail x} \qquad \textsc{Root-Subtree}\ \frac{\mathrm{subtree}(x) \in \Upsilon}{\Upsilon; \Gamma; L \vdash x \rightarrowtail x}$$

$$\textsc{Root-Alias}\ \frac{\Upsilon; \Gamma; L \vdash x \rightarrowtail p' \qquad p' \in \mathsf{MustAlias}(\cdot, p)}{\Upsilon; \Gamma; L \vdash x \rightarrowtail p}$$

$$\textsc{Root-Field}\ \frac{\Upsilon; \Gamma; L \vdash x \rightarrowtail p \qquad \Gamma; L \vdash p.f : \mathrm{complete}\langle y\rangle\ c}{\Upsilon; \Gamma; L \vdash x \rightarrowtail p.f}$$

$\boxed{\Gamma; L \vdash p : \tau}$

$$\textsc{T-Var-Storage}\ \frac{\Gamma(x) = q\ \tau}{\Gamma; L \vdash x : \tau}$$

$$\textsc{T-Field-Storage}\ \frac{\Gamma; L \vdash p : \tau_1 \qquad P \vdash \tau_2\ f \in \mathrm{Fields}(\mathrm{Class}(\tau_1)) \qquad \mathsf{LockedFinalAlias}(\Gamma, L, p, x)}{\Gamma; L \vdash p.f : \tau_2[x/\mathrm{this}]}$$

$$\textsc{T-Program}\ \frac{\mathrm{FieldsOnce}(\overline{class}) \qquad \mathrm{ClassesOnce}(\overline{class}) \qquad \forall c \in \overline{class}.\overline{class} \vdash c \qquad \emptyset; \emptyset; \emptyset \vdash e : \tau; \Upsilon'}{\vdash \overline{class}\ e}$$

$$\textsc{T-Class}\ \frac{\forall \tau\ f \in \mathrm{Fields}(c).\overline{class} \vdash \tau\ f}{\overline{class} \vdash c} \qquad \textsc{T-ValidPartialField}\ \frac{c \in \overline{class}}{\overline{class}; \Upsilon; \Gamma; L \vdash \mathrm{partial}\ c\ f}$$

$$\textsc{T-ValidCompleteField}\ \frac{c \in \overline{class}}{\overline{class} \vdash \mathrm{complete}\langle \mathrm{this}\rangle\ c\ f}$$

Figure 5. Supporting judgements and program typing

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
- 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

G	Operation	G'	Type Rule
$\mathrm{Forest}(G)$	Remove $X \to Y$	$\mathrm{Forest}(G')$, $\mathrm{Root}(Y)$, $X \| \| Y$	T-DFIELD
$\mathrm{Forest}(G)$, $\mathrm{Root}(Y)$, $X \| \| Y$	Add $X \to Y$	$\mathrm{Forest}(G')$	T-WFIELD-COMPLETE

Table 2. Transitions in a graph rewriting system on forests, relevant to ensuring the capability-granting relation remains acyclic. $\mathrm{Forest}(G)$ is the proposition that the graph G is a forest, $\mathrm{Root}(Y)$ is the proposition that Y has in-degree 0, and $X||Y$ is the proposition that X and Y are unreachable from each other.

kept acyclic. The complete system, including all shaded hypotheses, ensures deadlock freedom and incidentally prevents simultaneous access to fields. Allowing unsynchronized reads of partial references, or allowing unsynchronized reads of complete references to return partial references can be made sound because those operations would not allow use of stale lock group information, but omitting that flexibility simplifies presentation.

Figure 5 presents supporting judgements and auxiliary functions used in the main rules. This figure defines program typing, the relation $\Upsilon; \Gamma; L \vdash x \rightarrowtail p$ 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 for our core language. Figure 9 in the appendix has the additional runtime typing rules.

This section walks through each source typing rule in Figure 6:

- T-IF is purely structural. It types the branch condition, and then each branch with the tree assertions Υ_1 flowing out of the condition. It merges the output tree assertions of the branches conservatively, by set intersection.
- T-NULL simply types the constant null as a partial reference.
- T-SUB-TYPE admits a simple convenient subtyping.
- T-VAR is mostly straightforward, accounting for borrowing through the $\mathsf{Alias}(\tau)$ metafunction (Figure 5), but also a variable typed as borrowed can only be read if the borrowed reference is valid (must-aliased to a complete 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 actually a unique reference, rather than a borrowed reference.
- T-FIELD is the most basic field typing, which performs a normal (borrowing) read of a field. $\mathrm{FieldAccess}(\cdots)$ (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 $\mathrm{complete}\langle\mathrm{this}\rangle\ c$ field declaration for the current context. It ensures that evaluating the path will not change the capability-granting relation.
- T-WFIELD-COMPLETE, after checking the field access, checks for a final variable y aliased to the expression being stored ($\mathsf{MustAlias}$ 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-WFIELD-PARTIAL is similar to the previous rule, but simpler because there is no need to check for cycles. This rule does not use the $\mathsf{Alias}(\tau)$ function simply because it would be a no-op: borrowing a partial reference produces a partial reference.
- T-DFIELD handles destructive reads of complete 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 groupless, rather than complete, 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 complete 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 [16]).
- T-LET This is a mostly standard binding rule.
- 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 partial 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 can never violate any disjointness assumptions the spawning thread may have about the passed value. The language could be enriched to support assertion passing as well, but this rule is sufficient to show soundness of the overall approach.
- T-LOCK-FIRST, for acquiring a thread's first lock, is fairly straightforward: 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), and it requires that the resulting type is well-formed with respect to the current lock set. 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 $\mathsf{ValidCap}(\Gamma, L, p, y)$ 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 the "orphaned lock" criteria (Section 3.5) to ensure the thread does not lock the ancestor of an orphaned lock.

$\Upsilon := \{\text{root(Variable)} \mid \text{subtree(Variable)} \mid \text{Variable}||\text{Variable}\}$ set $\quad \Gamma : \text{Variable} \mapsto \text{Final Qualifier} * \text{Type} \quad L : \text{Variable list}$

$\boxed{\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'}$

$$\text{T-IF}\ \frac{\Upsilon; \Gamma; L \vdash e_1 : \tau_1; \Upsilon_1 \quad \Upsilon_1; \Gamma; L \vdash e_2 : \tau; \Upsilon_2 \quad \Upsilon_1; \Gamma; L \vdash e_3 : \tau; \Upsilon_3}{\Upsilon; \Gamma; L \vdash \text{if } e_1\ e_2\ e_3 : \tau; \Upsilon_2 \cap \Upsilon_3}$$

$$\text{T-NULL}\ \frac{}{\Upsilon; \Gamma; L \vdash \text{null} : \text{partial } c; \Upsilon}$$

$$\text{T-VAR}\ \frac{\Gamma(x) = q\ \tau \quad \tau' = \mathsf{Alias}(\tau) \quad \forall y \forall c.\tau' = \text{borrowed}\langle y\rangle\ c \Rightarrow \mathsf{ValidCap}(\Gamma, L, x, y)}{\Upsilon; \Gamma; L \vdash x : \tau'; \Upsilon}$$

$$\text{T-WVAR}\ \frac{\Gamma(x) = \text{nonfinal } \tau \quad \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'}{\Upsilon; \Gamma; L \vdash x := e : \mathsf{Alias}(\tau); \Upsilon'}$$

$$\text{T-DVAR}\ \frac{\Gamma(x) = \text{nonfinal guardless } c \quad \mathsf{FinalAlias}(\Gamma, x, y) \quad \text{root}(y) \in \Upsilon}{\Upsilon; \Gamma; L \vdash \text{dread}(x) : \Gamma(x); \Upsilon}$$

$$\text{T-FIELD}\ \frac{\tau_f = \mathsf{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad \tau = \mathsf{Alias}(\tau_f)}{\Upsilon; \Gamma; L \vdash p.f : \tau; \Upsilon}$$

$$\text{T-WFIELD-COMPLETE}\ \frac{\Upsilon; \Gamma; L \vdash e : \text{guardless } c; \Upsilon' \quad \text{complete}\langle x\rangle\ c = \mathsf{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad y \in \mathsf{MustAlias}(\cdot, e) \quad \text{root}(y) \in \Upsilon' \quad \Upsilon'; \Gamma; L \vdash r \rightarrowtail p \quad r||y \in \Upsilon'}{\Upsilon; \Gamma; L \vdash p.f := e : \text{borrowed}\langle x\rangle\ c; \Upsilon' /y}$$

$$\text{T-WFIELD-PARTIAL}\ \frac{\text{partial } c = \mathsf{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad \Upsilon; \Gamma; L \vdash e : \text{partial } c; \Upsilon'}{\Upsilon; \Gamma; L \vdash p.f := e : \text{partial } c; \Upsilon'}$$

$$\text{T-SUB-TYPE}\ \frac{\Upsilon; \Gamma; L \vdash e : \text{borrowed}\langle x\rangle\ c; \Upsilon'}{\Upsilon; \Gamma; L \vdash e : \text{partial } c; \Upsilon'}$$

$$\text{T-DFIELD}\ \frac{\mathsf{FinalAlias}(\Gamma, p.f, t) \quad t \notin \Upsilon \quad \text{complete}\langle x\rangle\ c = \mathsf{FieldAccess}(\Upsilon, \Gamma, L, p, f) \quad \Upsilon; \Gamma; L \vdash r \rightarrowtail p \quad \Upsilon' = \Upsilon \cup \{\text{root}(t)\} \cup \{t||z \mid \forall z.r||z \in \Upsilon'\}}{\Upsilon; \Gamma; L \vdash \text{dread}(p.f) : \text{guardless } c; \Upsilon'}$$

$$\text{T-NEW}\ \frac{\text{root}(z) \in \Upsilon \vee \text{subtree}(z) \in \Upsilon \quad \Upsilon_{\text{body}} = \Upsilon \cup \{\text{root}(y)\} \cup \{y||z\} \cup \{y||t \mid \forall t.t||z \in \Upsilon\} \quad \Upsilon_{\text{body}}; \Gamma[x \mapsto \text{guardless } c][y \mapsto \text{final partial } c]; L \vdash e : \tau; \Upsilon'}{\Upsilon; \Gamma; L \vdash \text{let guardless } x,\ \text{final } y = \text{new } c \text{ in } e : \tau; \Upsilon' /y}$$

$$\text{T-LET}\ \frac{\Upsilon; \Gamma; L \vdash e_1 : \tau_1; \Upsilon' \quad \Upsilon'; \Gamma[x \mapsto q\ \tau_1]; L \vdash e_2 : \tau; \Upsilon''}{\Upsilon; \Gamma; L \vdash \text{let } q\ x = e_1 \text{ in } e_2 : \tau; \Upsilon''/x}$$

$$\text{T-SPAWN}\ \frac{\Gamma(x) = \text{nonfinal } \alpha\ c \quad \emptyset; \{x \mapsto \text{final partial } c\}; \emptyset \vdash e : \tau; \Upsilon'}{\Upsilon; \Gamma; L \vdash \text{spawn } x\ e : \text{partial } c; \Upsilon}$$

$$\text{T-LOCK-FIRST}\ \frac{\Upsilon; \Gamma; [] \vdash p : \tau_1; \Upsilon \quad \mathsf{RaceFreePath}(\Gamma, [], p) \quad \mathsf{FinalAlias}(\Gamma, p, x) \quad \Upsilon_t = \mathsf{NewSubtrees}([], \Upsilon, x) \quad \neg\exists z.\text{subtree}(z) \in \Upsilon \quad \Upsilon \cup \Upsilon_t; \Gamma; [x] \vdash e : \tau; \Upsilon' \quad \forall y \forall c.\tau \neq \text{borrowed}\langle y\rangle\ c}{\Upsilon; \Gamma; [] \vdash \text{lock } p\ e : \tau; \Upsilon' /\{z \mid z \in \Upsilon_t\}}$$

$$\text{T-LOCK-N}\ \frac{\begin{array}{c}\Upsilon; \Gamma; L \vdash p : \tau_1; \Upsilon \\ \tau_1 = \text{borrowed}\langle y\rangle\ c \quad \mathsf{RaceFreePath}(\Gamma, L, p) \quad \mathsf{FinalAlias}(\Gamma, p, x) \quad \mathsf{ValidCap}(\Gamma, L, p, y) \quad \Upsilon; \Gamma; x :: L \vdash e : \tau; \Upsilon' \quad y \in L \\ \forall z \in L : L = L'@[z] \vee \exists p'.\mathsf{RaceFreePath}(\Gamma, L, p') \wedge \mathsf{FinalAlias}(\Gamma, p', z) \wedge \Gamma; L \vdash p' : \text{complete}\langle a\rangle\ c \quad \forall w \forall c.\tau = \text{borrowed}\langle w\rangle\ c \Rightarrow w \in L\end{array}}{\Upsilon; \Gamma; L \vdash \text{lock } p\ e : \tau; \Upsilon'}$$

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

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 [16] includes additional examples.

4.1 Opening Example

Section 1 gives an example of a very simple program that is surprisingly difficult to verify. Here we describe how the example type checks in our core language. We omit T2 and T4, which are similar to T1 and T3, respectively. Core language code for the threads is in Figure 7.

The typing derivation of Thread 1 is fairly simple. Assume $\Gamma = n2 \mapsto$ final partial Node, $L = []$, and $\Upsilon =$. The only rule permitting acquisition of locks when L is empty is T-Lock-First. The target lock expression type checks simply with the same input environment using T-Var. n2 is trivially a race free path and a final alias of itself. Υ_t will contain $\text{subtree}(n2)$ due to the NewSubtrees hypothesis of T-LOCK-FIRST, which will be used as the input Υ for the typing derivation of the critical section,

```
class Node { complete<this> Node left, right; }
Thread 1 : lock n2 { null }
Thread 3 :
1     lock n1 {
2       let final n1left = n1.left in
3       let final n1right = n1.right in
4       // Here, L = [n1]
5       // Υ = subtree(()n1)
6       lock n1.left {
7         // L = [n1left, n1]
8         lock n1.right {
9           // L = [n1right, n1left, n1]
10          null
11        }}}
```

Figure 7. The introduction's example, in our core language.

which is simply an application of T-NULL. The conclusion of T-LOCK-FIRST removes all assertions using n2 from the resulting Υ (because n2 was in Υ_t), and the output shape is empty.

Thread 3's typing derivation is more involved. Its initial environment is similar to thread 1: $\Gamma = n1 \mapsto$ final partial Node,

$L = []$, and $\Upsilon = .$ The base of the typing derivation is again T-LOCK-FIRST due to the similar input environment. The critical section on `n1` (lines 2-11) type checks in an environment with the same Γ, but an extended lock set and Υ. Υ picks up a subtree assertion as in the first thread, so the environment entering the critical section on line 2 is: $\Gamma = n1 \mapsto \text{final partial Node}$, $L = [n1]$, $\Upsilon = \text{subtree}(n1)$. The let expressions on lines 2 and 3 type check using T-Let, each adding a new final partial binding for the duration of their lexical scopes, which also adds must-alias facts for any reasonable must-aliasing analysis. The input context for line 6 is: $\Gamma = n1 \mapsto \text{final partial Node}, n1left \mapsto \text{final partial Node}, n1right \mapsto \text{final partial Node}$, $L = [n1]$, and $\Upsilon = \text{subtree}(n1)$. The acquisition of `n1.left` type checks using T-LOCK-N. The read of `n1.left` type checks using T-FIELD, which produces a value of type $\text{borrowed}\langle n1\rangle$ Node: the FieldAccess hypothesis internally types the variable access using T-VAR, ensures data race freedom, and binds the 'this' in the field type; and the Alias hypothesis converts the field's complete type to a borrowed type. Because that value is guarded by `n1`, the rule checks that `n1` is in L, which it is. The borrowed lock group information is still valid (ValidCap). To complete the use of T-LOCK-N to type the acquisition of `n1.left`, its critical section must also type check, in an environment with the same Γ and Υ, but the extended lock set $L = [n1left, n1]$ (entering line 8). The derivation for this inner acquire and critical section uses T-LOCK-N again, typing `n1.right` as $\text{borrowed}\langle n1\rangle$ Node, checking that `n1` is in L and other hypotheses, and deriving the type for the innermost critical section (null) using T-NULL in a further-extended environment, with the same Γ and Υ as on line 8, but with $L = [n1right, n1left, n1]$. Finishing the typing derivation for the innermost critical section, T-LOCK-N ensures that the type of the body (null) is not borrowed from `n1right`, keeping the type well-formed for the enclosing context. A similar restriction is enforced on the type exiting the T-LOCK-N derivation from lines 6-11. Each of the let expressions then removes its bound variable from the output context (enforcing lexical scoping), and finally the base of the derivation (T-LOCK-FIRST) removes all uses of `n1` from its output Υ.

4.2 Tree Rotations

Figure 1 implements the clockwise rotation operation in a splay tree. Splay trees are 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 [6], 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 partial references to nodes) are mutually disjoint after the destructive reads, and can therefore check that the capability-granting changes implied by storing the unique references back into the heap preserve the relation's acyclicity.

Figure 1's code shows how flexible the use of complete references for strong updates is: nowhere does the code need to explicitly state what the new guard on any lock is. Instead the changes to guards are entirely 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 [6, 19]. 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.

4.3 Arrays

Treated as a structure with integer-named fields containing complete 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 `complete Object[]` (an array of complete 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. Another possible approach is to lock cell referents in order of increasing index. Unfortunately, the relative ordering of dynamically-computed array indices is undecidable, so it is safe but difficult to verify.

4.4 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 [17]. 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 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 partial reference would still require no locks to be held at the call site.

4.5 Binary Tree Search

While searching in a basic binary tree does not take advantage of lock capabilities' flexibility, we show it to demonstrate that our approach does not add substantial complexity to simple examples. For brevity we extend the language and type system with basic support for integers: constants, assignment, comparison, and arithmetic.

```
class BTree {
  complete<this> left, right;
  int data;
  partial BTree find(int target)
    with <this>
  {
    // \Upsilon=subtree(this)
    // L=this::L' for abstract L'
    if (this.data == target) {
      this
    } else {
      if (this.data < target) {
        // Descend right
        if (this.right) {
          let final r = this.right in
          lock (this.right) {
            // Program point A
            this.right.find(target)
          }
        } else {
          null
        }
      } else {
        // Descend left
        if (this.left) {
          let final l = this.left in
          lock (this.left) {
            this.left.find(target)
          }
        } else {
          null
        }
      }
    }
  }
  groupless BTree removeLeftChild()
    with <this>
    produces root(result),result||this
  {
    // Upsilon=subtree(this)
    let final l = this.left in
    dread(this.left)
    // Upsilon=subtree(this),root(l),l||this
  }
}
```

The find method introduces a basic method summary specifying which capabilities (and thus, locks) the method requires to execute. The clause `with <this>` specifies that the the capability associated with the 'this' instance must be provided by the caller (caller synchronization). If an arguments' capability is required (the method assumes an argument is locked on method entry) it can be referred to by the parameter's formal name (e.g. `with <this>, <arg>`). At the call site, the type system substitutes the actual arguments in for the formal capability names in the method summary, and checks the instantiated capability requirements against the locks held. Note that to satisfy a capability requirement at a call site, a final alias to the required lock must already exist in the caller's frame if the capabilities match, since the required lock is in L.

At program point A, the static lock set will be $L = r :: \text{this} :: L'$, where L' is an abstract (possibly empty) list of other locks that may be held at the call site (lock set polymorphism). The fact that L contains `r` satisfies the `with <this>` clause of the method signature for the recursive call. When checking the call site, the `this` in the signature is replaced by a final alias of the method dispatch target (`r`), and that is checked against the set of locks held at the call site. The presence of additional locks in L is sound because the call does not assume an empty lock set.

The `removeLeftChild` method serves limited practical purpose but demonstrates how Υ is treated in method summaries. Methods may use an `assume` clause to specify a partial input Υ, and a `produces` clause to specify a partial output Υ. $\text{subtree}(\text{this})$ is implicitly present for the input Υ. All Υs may specify roots and disjointness assertions. The output Υ may refer to the special variable `result` referring to the return value, and only if the return type is guardless. The input Υ may specify a new assertion $\text{local}(x)$ that behaves similarly to $\text{subtree}(x)$ by providing well-formedness constraints on Υ: giving a non-root variable relative to which to specify disjointness. At call sites, the type system requires that the roots of the capability-granting trees such $\text{local}(x)$ assertions apply to satisfy the disjointness constraints applied to those local anchors.

4.6 Counting Non-Zero Tree Elements

Another simple example to show that basic tasks are not made more difficult by our system:

```
class Tree {
  complete<this> left, right;
  int data;
  int CountNonZero()
    with <this>
  {
    let final l = this.left in
    let final r = this.right in
    let final nleft = if (this.left) {
                        lock (this.left) {
                          this.left.CountNonZero()
                        }
                      } else { 0 } in
    let final nright = if (this.right) {
                         lock (this.right) {
                           this.right.CountNonZero()
                         }
                       } else { 0 } in
    let final nthis = if (this.data == 0) { 0 } else { 1 } in
    (nleft + nright + nthis)
  }
}
```

4.7 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 class parameterization as in RCC/JAVA [13] to refer to external locks (another use of final variables for lightweight singleton types), along with a notion of *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 8. 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 partial 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 perma-

```
class CircularList {
  fixed<this> CircularListNode<this> head;
}
class CircularListNode<ghost CircularList list> {
  fixed<list> CircularListNode<list> prev;
  fixed<list> CircularListNode<list> next;
  complete<this> Object data;
  public fixed<l> CircularListNode<l> find(partial 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(
    partial 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 8. A circular list, using fixed-guard references and external capabilities.

nent capability-granting relationships unrelated to heap structure, several invariants would need slight adjustments.

4.8 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* [6, 12, 19]: 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.

4.9 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, having a whole-problem lock guarding all "chopstick" locks. Allowing early releases would still serialize the acquisition of multiple locks, but not their holding (which structured locking does serialize, as in the circular list). Explicit early releases would allow parallelism between multiple threads that each hold multiple locks; once a thread acquired the necessary "chopstick" locks, it could release the whole-problem lock, and continue using the chopstick-lock-protected resources while other threads acquired the whole-problem lock and their chopsticks.

5. Soundness

We have proven that our type system ensures deadlock freedom. The full proof is in the appendices. 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 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 ϕs 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.

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 no longer so direct. But 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 some other 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 4.8), 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 a number of related areas, including prior approaches to static deadlock detection and work on linear and unique types. We begin with a discussion of the dominant approach to statically verifying deadlock freedom — *lock levels* — and then discuss other related work.

6.1 Lock Levels

The most common and well-established approach to statically verifying deadlock freedom is a family of techniques called *lock levels* [6, 12, 18–21, 26]. The heart of the approach is a static partitioning of locks into levels, and a fixed partial ordering on those partitions. Then a verifier checks that any time a thread acquires a lock it either holds no locks, or the target lock's level is ordered after the level of *every* lock it already holds.

[2] We 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 lock levels verification approach typically places two serious limitations on code:

- **Total Ordering**: It must be possible to establish a total ordering on any set of locks that are held concurrently.
- **Fixed Ordering**: It is not possible to dynamically change the relative acquisition order of two or more locks. This restriction has two components:
  - **Partition Ordering**: It is not possible to change the ordering of partitions dynamically.
  - **Partition Membership**: It is not possible to move locks among partitions.

Lock levels are adequate for coarse-grained lock ordering, such as between layered subsystems of a program. However, these restrictions are limiting for dynamic data structures using fine-grained locking.

The **total ordering restriction** is problematic in any structure where there is no natural ordering on the locks acquired. Our circular OS process list example is a great demonstration of this: because there is no sensible way to totally order the list nodes so any thread can acquire multiple locks, the list cannot be expressed in a lock levels system. Because process locks can be acquired in any order by a thread holding the guarding capability, making the process list nodes all children of the whole-list lock makes it possible to verify safety of this locking discipline using lock capabilities.

Lock capabilities' acquisition order flexibility also extends beyond "siblings" in the capability-granting relation. A thread holding a lock could lock a grandchild before a child (such as `btree.left.left` before `btree.right`), assuming the (different) child guarding the grandchild (e.g., `btree.left`) was locked appropriately in advance. In (non-parameterized) lock level systems, this would butt heads with the total ordering restriction. In some parameterized lock levels systems, it would be possible to fix an order on the children for cases like locking both children of a binary tree node by declaring that for a node in level n, the left child would be in level $n + 1$, the right in $n + 2$, etc. to establish a total ordering on the nodes acquired. But as additional layers of a structure need to be accessed as well, spacing the relative levels of the children appropriately becomes increasingly difficult and unnatural. In the capability-based view, if the parent node grants the capability for both children, this style of locking behavior is entirely natural.

The **fixed ordering** restriction causes problems in algorithms that change lock orders dynamically, such as rebalancing binary trees, or even removing a node from a binary tree. Recall that there are two sub-restrictions that combine to produce the fixed ordering restriction: the **partition ordering** restriction, and the **partition membership** restriction. Lifting either of those sub-restrictions is sufficient to allow changing the relative ordering of locks. For example, in a system with only one of these restrictions, it would be possible to verify deadlock freedom for code reversing the list-order of a fine-grained linked list, because either the partitions for each list node could be reordered, or the nodes themselves could be moved between partitions. To the best of our knowledge no lock level system permits partition reordering. So the determining factor is the presence or absence of the partition membership restriction.

Lifting the partition membership restriction requires control of aliasing to ensure no reference to a moved object has an out-of-date level for its referent. To the best of our knowledge, the partition membership restriction is present in all but two pieces of lock levels work: SAFEJAVA [6], which uses ad hoc extensions and an unspecified flow-sensitive analysis dealing with reachability and aliasing; and CHALICE [19–21], which uses fractional permissions [7] to

strictly control aliasing of a ghost variable that defines an object's lock level.

Boyapati et al.'s SAFEJAVA [6] extends lock levels with simple tree-based orderings of locks within a level. A thread in SAFEJAVA is permitted to acquire a lock when it holds no locks, holds only locks in levels ordered before the target lock, or the target lock is a tree-descendent of the most recently acquired lock. This permits more flexibility for fine-grained locking than basic lock levels, but still suffers from the total ordering restriction. SAFEJAVA also supports reordering within a tree, but uses an unspecified analysis pass to maintain the tree invariant. Our system can in some sense be seen as a generalization of SAFEJAVA's tree-based ordering, extending past trees and exploiting exclusivity of the parent lock to achieve greater flexibility. We also fully specify the analysis to maintain the capability-granting relation's forest property, while Boyapati et al. provide only a basic intuition for how their analysis works. SAFEJAVA's use of the tree ordering within a lock level is the inspiration for the lock capability embedding described in section 4.8. SAFEJAVA also includes intra-level ordering based on fixed DAGs between locks, which capabilities could be extended to support, and a separate class of unordered locks with a new locking primitive to acquire n locks at once by internally providing a total ordering on the locks (e.g. memory order).

Leino et al. describe CHALICE, a system using a novel variation on lock levels to avoid deadlock [19, 20]. 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 [7] 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 [7] or using a counting permission type system [3] to track propagation. Bringing some of inspiration from CHALICE to bear on lock capabilities, we could apply fractional permissions or counting permissions [3] to the guard portion of complete references, allowing controlled sharing of guards while still permitting strong updates to the guards. These sharing approaches could also lay the groundwork for supporting a SAFEJAVA-style DAG ordering, but with mutation.

With lock capabilities, changing the relation of which locks grant capabilities to acquire other locks effectively breaks both the **partition ordering** and **partition membership** sub-restrictions of lock levels: reversing the capability-granting relationship between two locks reverses the possible acquisition order of locks they guard, and to be able to change that relationship requires being able to associate the original "parent" capability-granting lock with the capability of its former child. We also lift the total ordering restriction on lock acquisition, the cost of which is the need to handle the orphaned lock problem. Systems with the total ordering restriction trivially avoid orphaned lock issues because any capability that is unsafe to use (because it precedes an orphaned lock in the capability granting relation) comes from a lock that is not "ordered after" the orphaned lock itself.

There has also been work using similar terminology to ours for techniques to avoid deadlock in message passing code [18], including an extension to CHALICE [21]. These systems enforce a partial order similar to lock levels on channel sends, using an additional system of *obligations* to ensure that threads do not block waiting to receive on a channel unless some other thread holds an obligation to send a message on that channel. This is also similar to an approach for adapting deadlock-freedom techniques to unstructured primitives [26], where a thread that acquires a lock also picks up a statically enforced obligation to release the lock later.

Table 3 summarizes the limitations of lock levels, along with important examples that run afoul of those limitations. Lock capabilities suffer from none of the problematic limitations we listed for lock levels.

6.2 Other Static Techniques for Deadlock Freedom

Attiya et al. [2] describe core results about the range of locking protocols for fully encapsulated data structure implementations that can be verified as deadlock-free and atomic. Their results include the DYNAMIC TREE LOCKING protocol, which is similar to the locking protocol described here, but there are crucial differences. First, their technique is only applicable to strongly encapsulated implementations of data structures where the heap structure is the only control of lock ordering, and where encapsulated locks are globally unique when a data structure is not under modification. Thus they can verify strongly encapsulated binary tree implementations, but could not verify the circular list without requiring references from the global list lock to every node. Attiya's distinction between external locks visible outside the module, and internal locks that are globally unique (have only one reference in the program) is also important: this also prevents verification of the circular list example as it is used in OS kernels, where all process object locks are visible throughout the kernel. We make no distinction between global or module-local locks. Part of what allows our approach to handle the circular list example is that we crucially distinguish heap-directed capability-granting edges from fixed, heap-agnostic edges in the capability-granting relation, while they conflate safe acquisition order with heap structure. Their protocol is also described in axiomatic terms, though they briefly describe experiences with an analysis implemented in an analysis framework; we have provided a concrete description of how to enforce a safe locking protocol, but have yet to implement it. The fact that they guarantee atomicity of each module operation is also important, and results in very different expressiveness between our systems even within a module: their locking protocols do not allow threads to reacquire locks released since a thread's first acquisition. Thus they do not need to address the problems with orphaned locks. This also prevents their technique from generalizing to condition variables, while ours does so naturally (with further work needed to prevent lost wake-ups). Our deadlock-freedom type system could be coupled with a separate atomicity analysis. If they sacrificed atomicity by allowing reacquisitions (when it would not cause deadlock), they would still have no problems with orphaned locks, because the encapsulation assumption forces any other threads to start locking from exposed locks, preventing other threads from holding the recently-released descendant of the thread's first lock, at the cost of increased sharing.

Attiya et al.'s work grew out of work from the database community [9] on locking protocols for locking in databases of directed graphs. This work focused entirely on what would be safe and serializable, axiomatically, in a database under the assumption that locking protocols would be enforced by a central concurrency management system.

Limitation	Unverifiable Example	Reason Lock Levels Cannot Verify
Total Ordering	OS kernel circular list	No total order on list nodes
	Trees	Specifying a total order for all but the simplest example is difficult
	Array-order locking	Ordering on dynamically computed indices is generally undecidable
Partition Ordering	Reversing a	No mechanism to reorder partitions
Partition Membership	fine-grained linked list	Requires strong updates / aliasing control

Table 3. Examples that cannot be checked by lock levels due to that technique's limitations, and why the examples violate the corresponding restriction. Each example can be checked by a lock capabilities system because lock capabilities do not suffer from the problematic restrictions.

Wang et al. [30] describe a way of both verifying deadlock freedom and synthesizing logic to dynamically avoid deadlock at runtime. Their analysis converts the control flow graph of a target program into a Petri net of a special shape, where the property of allowing deadlock can be expressed in what is a standard form for properties on Petri nets, for which there are known algorithms to enrich a Petri net to avoid such a property. Such an enriched petri net is then translated back to additional instrumentation that avoids deadlock dynamically. Some of the instrumentation they generate strongly resembles the locking protocol for lock capabilities; for example, they automatically synthesize solutions equivalent to our array access solution [29]. However, the results of the process can be unpredictable. The instrumentation synthesis has generated low-overhead code in experiments, but there is no guarantee of good behavior. Because the synthesis process is iterative, each iteration adding new locks to avoid deadlocks present in the previous iteration, many layers of locks could be introduced where a programmer could explicitly insert fewer, and have their use verified as correct using lock capabilities. Their approach is also highly sensitive to resolving non-aliasing between locks: their tool tends to generate per-type locks to acquire before any lock of each type, because it cannot differentiate unrelated locks for the same type. Their tool could be enriched with an aliasing analysis, but even that would have difficulty with recursive structures like binary trees. Our technique gains precision from using complete references for such structures, and in conjunction with lock levels can naturally separate uses of the same basic structure in unrelated parts of a program.

6.2.1 Hybrid Static/Dynamic Deadlock Prevention

Recently there has been work on statically ensuring deadlock freedom with the aid of an enriched lock acquisition primitive, which does not require fixing any lock order. This is also called deadlock *avoidance* due to the dynamic component to ensuring deadlock freedom. Boudol [4] and Gerakios et al. [14, 15] describe type and effect systems that inform an extended locking primitive about which locks a thread will acquire before releasing the current target lock. This extended locking primitive then does not try to acquire its target lock unless all locks that *will be acquired* before releasing the target lock are also available. This requires a locking primitive taking time at best linear in the number of locks that will be held concurrently.

Gerakios et al.'s benchmarks are a naïve implementation of the dining philosophers problem (yielding an expected performance gain from improved parallelism over the deadlock-free execution with standard primitives[3]); and several simple benchmarks showing no performance penalty, some with only one lock, several with two locks, and one with more than two locks, but where no more than two are ever acquired concurrently. These do not explore the effects of the lock primitive on deep locking paths as sometimes occur in highly concurrent software. Additionally, their benchmarks other than the dining philosophers use only currently-working code modified to use their primitive; these programs have total orderings on locks held simultaneously, which means that acquisitions of n locks are likely to often fail checking one of the first couple locks anyways, because all threads acquire locks in the same order. Programs written without assuming any lock ordering and which acquire more than two locks at once are likely to have very different performance characteristics for this primitive than those explored.

It is also not clear how these systems interact with verification of race freedom. If the target of a future lock acquisition depends on the content of memory protected by an earlier acquisition, checking that lock's state during the earlier acquisition requires a data race; not only do the semantics for the locking construct require a data race, but if the locking primitive's check of the racily-identified lock succeeds before another thread correctly changes the lock the relevant field points to, it appears that the approach becomes unsound (the core language for which they prove soundness is not capable of expressing this situation). This would cause problems verifying examples such as fine-grained binary trees. One version of Gerakios et al.'s system [14] also ensures data race freedom, but the reference typing is so restrictive that the type of a mutable reference to a lock uniquely identifies the only lock that could ever be referenced by that cell (through singleton types that cannot be abstracted). Gerakios et al. have an implementation of this deadlock-free system [15], which must therefore either race and be unsound, or conservatively over-approximate the locks read from memory using the results of their pointer analysis. The latter would make acquiring multiple locks in a structure like a recursive binary tree very expensive.

These orderless-locking systems have an additional penalty, which is that separate compilation is not supported, and it is not clear how to extend the systems. Essentially, in this system a function type τ_1 is a subtype of function type τ_2 only if τ_1 includes fewer lock acquisitions. Consider for example, swapping out a coarse-grained storage library for a fine-grained one with the same interface, in a program that holds locks while using the storage library. The fine-grained library acquires more locks internally, so this change would require re-type-checking and recompiling the whole program to generate code specifying additional future acquisitions for many lock primitives. Lock levels support modularity better through adding lock levels that do not interfere with other code, and lock capabilities support modularity better by using relatively local properties. Put another way, in orderless-locking systems the typing of an individual lock acquisition depends on both the context in which it occurs *and* the body of the lock statement[4], while for lock levels and lock capabilities the typing depends only on the outer context, requiring only that the body is well-typed, not the particulars of its typing.

[3] Note that in a system with unstructured locking using standard primitives, the best deadlock free solution could be verified by an unstructured adaptation of lock capabilities: each stick lock is guarded by a single global lock, and philosophers eat by acquiring the global lock, acquiring the two needed stick locks, releasing the global lock, and eating.

[4] Roughly; Gerakios et al.'s system supports unstructured locking.

6.3 Linear/Unique Types, and Strong Updates

There has been a long line of work on linear and unique types in programming languages, much of which permits forms of temporary aliasing known as borrowing [8, 11, 17]. A core thread that runs through nearly all prior work on unique types is the assumption that the common case is a single reference, and any aliasing is temporary and strictly controlled. This is a crucial assumption for the typical motivation for unique and linear types, which is resource isolation to make operations like memory deallocation safe.

Another use of linearity has been to permit strong updates to types. Alias types [25, 27] allow unique type assertions to be used for strong updates, or for the assertions to be made non-linear, after which no further strong updates are permitted to that object.

Our use of *partial uniqueness* can weaken uniqueness in a different manner because the motivation for using uniqueness does not require (or even desire) the ability to recollect all references to an object at one program point; we require strict control of only a single part of an object's type. Thus we permit arbitrary aliasing at a partial type, but maintain uniqueness of the reference that defines which lock grants the capability to acquire an object, to allow strong updates to only the portion of an object's type isolated to its complete reference.

CHALICE [19–21] uses fractional permissions [7] to control access and updates to ghost fields of objects, where those ghost fields define the level of a lock within a lock level system. The permissions guard field individually, which allows CHALICE to express structures where usable references to locks are shared, but the ordering information is restricted to one or a few known locations. This is similar to our use of partial uniqueness, and lock capabilities could be adapted to use fractional permissions to permit more sharing of an object's guard information without resorting to fixed references (Section 4.7) and losing the ability to change the lock's guard.

Müller and Rudich describe a system of ownership transfer in Universe Types with Transfer (UTT) [23] which resembles our use of strong updates to change the guard of a lock. One key difference is the granularity of the update: UTT transfers ownerships of groups of objects, rather than of individual objects, a byproduct of the fact that it supports multiple references carrying the same owner by using *external uniqueness* [17], which can be duplicated within a module but not shared outside of it (for example, allowing a doubly-linked list). Because we use total uniqueness we cannot express some of the same examples. The flip side to the external vs. total uniqueness choice is that while UTT's ownership transfer is union-based (transfer unions the owned sets; a list's nodes cannot be transferred into another list then back out without taking the other list's original nodes with them), our partially-unique references support changing the guard of individual locks as much as desired, moving locks into and out of lock groups almost arbitrarily (subject to checks to ensure an acyclic capability granting relationship). Our mechanism is also a very simple extension to basic unique references; while we only use partial uniqueness here for part of a type that corresponds closely to a tree-based ordering, the mechanism could just as easily be applied to share parts of a data structure, e.g. partial references of the form $x : \text{ref}\ (int*\_*float)$, which has possible security uses for modules of differing levels of trust whose type safety can still be verified.

6.4 Static Data Race Prevention

There has been a substantial amount of work on type systems for static race freedom [5, 6, 12, 13], and even some hybrid static/dynamic approaches [10]. A full survey is beyond the scope of this paper, but perhaps the best-known is RCC/JAVA [1], from which we borrow the technique of using final variable names for lightweight singleton types (or in our case, singleton lock groups).

6.5 Capability Systems

There has been a variety of work on capability type systems, mostly focused on memory safety. Walker, Crary, and Morrisett describe a capability calculus to control safe memory use and deallocation [28]. Smith, Walker, and Morrisett describe *alias types*, a type system for typed assembly language that ensures memory safety while allowing strong updates for operations such as object initialization [25, 27]. These systems are focused on ensuring that code accessing some memory cannot do so without a capability that is unavailable after that memory is released. Our system ensures code cannot acquire more than one lock without capabilities that become unavailable after releasing the granting locks.

6.6 Aliasing Analyses

A full overview of aliasing analyses is beyond the scope of this paper. However, Might et al. [22] and Smaragdakis et al. [24] provide good overviews of the precision of current state-of-the-art pointer analyses. We believe the precision necessary for resolving aliasing in our type system is met by recent work; we only require intraprocedual aliasing analysis for recent local assignments and destructive reads among local variables and fields of locked objects. Alternatively, a treatment of aliasing could be baked into the type system by adapting techniques like alias types [25, 27], as an early version of our system did.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for Safe Locking: Static Race Detection for Java. *ACM TOPLAS*, 28, March 2006.

[2] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential Verification of Serializability. In *POPL*, 2010.

[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 and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[7] J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, 2003.

[8] J. T. Boyland and W. Retert. Connecting Effects and Uniqueness with Adoption. In *POPL*, 2005.

[9] V. K. Chaudhri and V. Hadzilacos. Safe Locking Policies for Dynamic Databases. In *PODS*, 1995.

[10] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP*, September 2007.

[11] M. Fähndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, 2002.

[12] C. Flanagan and M. Abadi. Types for Safe Locking. In *ESOP*, 1999.

[13] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.

[14] 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.

[15] P. Gerakios, N. Papaspyrou, and K. Sagonas. A Type and Effect System for Deadlock Avoidance in Low-level Languages. In *TLDI*, 2011.

[16] 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.

[17] P. Haller and M. Odersky. Capabilities for Uniqueness and Borrowing. In *ECOOP*, 2010.

$\Sigma : \text{Location} \mapsto \text{Class} \qquad \phi : \text{Value} \mapsto \text{Variable}$

$\boxed{\Sigma; \phi; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon' \text{ cont.}}$

T-ANY-LOCATION
$$\frac{\Sigma(l) = c}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \text{partial } c; \Upsilon}$$

T-UNIQUE-NULL
$$\frac{\Gamma(y) = \text{final } \tau \qquad \text{null} \in \mathsf{MustAlias}(\cdot, y) \qquad \text{root}(y) \in \Upsilon}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{null} : \text{guardless } c; \Upsilon}$$

T-UNIQUE-LOC
$$\frac{\Gamma(y) = \text{final } \tau \qquad l \in \mathsf{MustAlias}(\cdot, y) \qquad \text{root}(y) \in \Upsilon}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash l^\bullet : \text{guardless } c; \Upsilon}$$

T-BORROWED-VALUE
$$\frac{\phi(v) = x \qquad p \in \mathsf{MustAlias}(\cdot, v) \qquad \Gamma; L \vdash p : \text{complete}\langle x \rangle\ c}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \text{borrowed}\langle x \rangle\ c; \Upsilon}$$

T-WITHLOCK
$$\frac{\begin{array}{c}\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \tau_1; \Upsilon_1 \\ \mathsf{FinalAlias}(\Gamma, l, x) \qquad \Upsilon_t = \mathsf{NewSubtrees}(L, \Upsilon, x) \\ \forall z.\text{subtree}(z) \in \Upsilon_t \Rightarrow \text{subtree}(z) \in \Upsilon_1 \\ \Sigma; \phi; \Upsilon_1; \Gamma; x :: L \vdash e : \tau_2; \Upsilon_2 \qquad \tau_2 = \text{borrowed}\langle y \rangle\ c \Rightarrow y \in L\end{array}}{\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{withlock } l\ e : \tau_2; \Upsilon_2 \ / \{z | z \in \Upsilon_t\}}$$

Figure 9. Runtime typing. These rules supplement runtime equivalents of the source typing rules from Figures 6 and 5, which are mostly the same rules with the additional Σ and ϕ static contexts, unused. The exception is the runtime version of T-SPAWN, which uses an empty ϕ to type check its body. Σ maps dynamic locations to class types. ϕ associates values (locations and null) with lock groups.

[18] N. Kobayashi. A New Type System for Deadlock-Free Processes. In *CONCUR*, August 2006.

[19] K. R. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *ESOP*, 2009.

[20] K. R. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, 2009.

[21] K. R. Leino, P. Müller, and J. Smans. Deadlock-free Channels and Locks. In *ESOP*, March 2010.

[22] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis. In *PLDI*, 2010.

[23] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *OOPSLA*, 2007.

[24] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *POPL*, 2011.

[25] F. Smith, D. Walker, and J. G. Morrisett. Alias Types. In *ESOP*, 2000.

[26] K. Suenaga. Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. In *APLAS*, 2008.

[27] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *TIC*, 2000.

[28] D. Walker, K. Crary, and G. Morrisett. Typed Memory Management via Static Capabilities. *ACM TOPLAS*, 22, July 2000.

[29] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI*, 2008.

[30] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL*, 2009.

A. Type Preservation

A.1 A Note on Static Must-Alias Analyses

In this proof, we treat the must-alias analysis as if it has oracle semantics. That is, in any dynamic context, $p \in \mathsf{MustAlias}(\cdot, p') \Leftrightarrow V|H(p) = V|H(p')$ (the analysis says two paths are aliased if and only if they are at that runtime execution point). This simplifies the proof by making it easier to derive must-alias facts when needed from equalities implied by other hypotheses.

This does not complicate the proof that type checking under the source type system implies successful checking under the runtime system, or make the system impractical. Any must-alias facts that can be proven using a sound static must-alias analysis can also be proven by our oracle must-alias analysis because the type rules never negate the results of the alias analysis.

A.2 Type Preservation Proof

Lemma 1 (Value Effects)**.** *If $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$ and e is a value, then $\Upsilon = \Upsilon'$.*

Proof. By induction on the derivation of $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$. □

Lemma 2 (Environment Strengthening)**.** *If $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$ then*

1. *for $l \notin \text{Dom}(\Sigma)$, $\Sigma, l \mapsto c; \phi; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$*
2. *for $l \notin \text{Dom}(\phi)$, x, $\Sigma; \phi[l \mapsto x]; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$*
3. *for all x, $\Sigma; \phi[\text{null} \mapsto x]; \Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$*
4. *for $[x||y] \notin \Upsilon$, $\Sigma; \phi; \Upsilon, x||y; \Gamma; L \vdash e : \tau; \Upsilon''$ where $\Upsilon'' \subseteq \Upsilon, x||y$*
5. *for $[\text{root}(x)] \notin \Upsilon$, $\Sigma; \phi; \Upsilon, \text{root}(x); \Gamma; L \vdash e : \tau; \Upsilon''$ where $\Upsilon'' \subseteq \Upsilon, \text{root}(x)$*
6. *for $x \notin \text{Dom}(\Gamma)$, $\Sigma; \phi; \Upsilon; \Gamma, x \mapsto \tau; L \vdash e : \tau; \Upsilon'$*

and

7. *If $\Sigma \vdash V : \Gamma$, then for $l \notin \text{Dom}(\Sigma)$, $\Sigma, l \mapsto c \vdash V : \Gamma$*

Proof. By induction on each derivation. □

Lemma 3 (Thread Type Preservation)**.** *If*

- *H1.* $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E[e] : \tau; \Upsilon'$ *(the thread's expression type checks)*
- *H2.* $H, i, V, Ls, e \rightarrow H', V', Ls', e'$ *(the thread performs a local reduction)*
- *H3.* $Ls = Ls_E @ Ls_L$ *(with the next two hypotheses, the dynamic lock set is partitionable into those corresponding to the static lock set L and those matching withlock expressions in $E[e]$)*
- *H4.* $\Gamma; V \vdash Ls_L : L$
- *H5.* $Ls_E \vdash E[e]$
- *H6.* $\vdash P$ *(the program is well-typed)*
- *H7.* $\vdash H : \Sigma$ *(the heap is well-typed)*
- *H8.* $\phi; \Gamma; Ls \vdash H$ *(ϕ reflects only the capability grants for the locks held)*
- *H9.* $H; V \vdash \Upsilon$ *(the heap and local variables are accurately modeled by Υ)*
- *H10.* $\Sigma \vdash V : \Gamma$ *(local variables are typed correctly)*
- *H11.* $H \vdash_i Ls$ *(the heap respects the reduced thread's lock set)*
- *H12.* $\Gamma \vdash \Upsilon$ *(Υ is well-formed w.r.t. Γ; it references only final variables)*
- *H13.* $\mathsf{CompleteForest}(H)$ *(the heap reflects a forest-shaped capability granting relation)*
- *H14.* $\mathsf{ValidUniques}(H, (i, V, Ls, e), \Sigma, (\phi, \Upsilon, \Gamma))$ *(each object this thread might access has at most one unique reference)*

$$TT : tid \to (\phi, \Upsilon, \Gamma) \qquad \phi : l \mapsto \langle x \rangle$$

$\boxed{Ls \vdash E[e]}$

CS-NO-MORE
$$\frac{E[e] \text{ doesn't contain withlock}}{[] \vdash E[e]}$$

CS-RECUR
$$\frac{E \neq \text{withlock} \qquad Ls \vdash E'[e] \qquad E = \ldots E'[e] \ldots}{Ls \vdash E[e]}$$

CS-LOCK-HELD
$$\frac{Ls \vdash E[e]}{Ls@[l] \vdash \text{withlock}\, l\, E[e]}$$

$\boxed{\phi; \Gamma; Ls \vdash H}$

T-VALID-CAP-ORACLE
$$\frac{\begin{array}{c}\phi(v) = \langle x \rangle \Leftrightarrow \exists f, l' \in Ls.\mathsf{FinalAlias}(\Gamma, l', x) \wedge H(l') = \langle c, F, o \rangle \wedge P \vdash \text{complete}\langle \text{this} \rangle\, c_f\, f \in \text{Fields}(c) \wedge F(f) = v \\ v \in \text{Dom}(\phi) \Rightarrow v = \text{null} \vee v = l^\bullet\end{array}}{\phi; \Gamma; Ls \vdash H}$$

$\boxed{H; V \vdash \Upsilon}$

T-VALID-TREE-ASSERTIONS
$$\frac{\begin{array}{c}\text{root}(x) \in \Upsilon \Rightarrow \neg\exists l, f.H(l) = \langle c, F, o \rangle \wedge P \vdash \text{complete}\langle \text{this} \rangle\, c_f\, f \in \text{Fields}(c) \wedge \mathsf{Dup}(F(f)) = \mathsf{Dup}(V(x)) \\ x||y \in \Upsilon \Rightarrow \neg\exists f_1 \ldots f_n.V(y) = l_1 \wedge \forall i > 1 : \left(\begin{array}{l} H(l_{i-1}) = \langle c_{i-1}, F_{i-1}, o_{i-1} \rangle \wedge \\ P \vdash \text{complete}\langle \text{this} \rangle\, c_i\, f_{i-1} \in \text{Fields}(c_{i-1}) \wedge \\ l_i^\bullet = F_{i-1}(f_{i-1}) \end{array} \right) \wedge l_n = \mathsf{Dup}(V(x)) \\ \text{subtree}(x) \in \Upsilon \wedge \text{subtree}(y) \in \Upsilon \Rightarrow x = y \qquad x \in \Upsilon \wedge y \in \Upsilon \Rightarrow x = y \vee V(x) \neq V(y) \vee \text{subtree}(x) \vee \text{subtree}(y) \\ x \in \Upsilon \Rightarrow \text{root}(x) \in \Upsilon \vee \text{subtree}(x) \in \Upsilon \qquad x||y \in \Upsilon \Rightarrow \text{root}(x) \in \Upsilon \vee \text{root}(y) \in \Upsilon \\ \text{root}(x) \in \Upsilon \wedge \text{subtree}(y) \in \Upsilon \Rightarrow x \neq y \qquad x \in \Upsilon \wedge y \in \Upsilon \wedge V(x) = V(y) \Rightarrow \neg\exists z_1 \ldots z_n.(\forall i \in (1..n-1).z_i||z_{i+1} \in \Upsilon) \wedge z_1 = x \wedge z_n = y\end{array}}{H; V \vdash \Upsilon}$$

$\boxed{\Sigma \vdash V : \Gamma}$

T-VALID-LOCAL-TYPES
$$\frac{V(x) = l \Rightarrow \Gamma(x) = q\, \alpha\, c \wedge \Sigma(l) = c \qquad \Gamma(x) = q\, \text{guardless}\, c \Rightarrow V(x) = \text{null} \vee V(x) = l^\bullet \qquad V(x) = l^\bullet \Rightarrow \Gamma(x) = q\, \text{guardless}\, c}{\Sigma \vdash V : \Gamma}$$

$\boxed{H \vdash_i Ls}$

T-LOCKS-HELD
$$\frac{\forall l \in Ls : H(l) = \langle c_l, F_l, \text{Some}(i) \rangle}{H \vdash_i Ls}$$

$\boxed{\Gamma \vdash \Upsilon}$

T-FINAL-VAR-TREES
$$\frac{\text{root}(x) \in \Upsilon \Rightarrow \Gamma(x) = \text{final}\, \tau \qquad \text{subtree}(x) \in \Upsilon \Rightarrow \Gamma(x) = \text{final}\, \tau \qquad x||y \in \Upsilon \Rightarrow \Gamma(x) = \text{final}\, \tau \wedge \Gamma(y) = \text{final}\, \tau}{\Gamma \vdash \Upsilon}$$

$\boxed{\vdash H : \Sigma}$

T-HEAP
$$\frac{\begin{array}{c}H(l) = \langle c, F, o \rangle \Leftrightarrow \Sigma(l) = c \qquad H(l) = \langle c, F, o \rangle \Rightarrow F(f) = l_f \Rightarrow P \vdash \alpha_f\, c_f\, f \in \text{Fields}(c) \wedge \Sigma(l_f) = c_f \\ H(l) = \langle c, F, o \rangle \Rightarrow F(f) = l_f^\bullet \Rightarrow P \vdash \text{guardless}\, c_f\, f \in \text{Fields}(c) \wedge \Sigma(l_f) = c_f \\ H(l) = \langle c, F, o \rangle \Rightarrow P \vdash \text{guardless}\, c_f\, f \in \text{Fields}(c) \Rightarrow (F(f) = l_f^\bullet \wedge \Sigma(l_f) = c_f) \vee F(f) = \text{null}\end{array}}{\vdash H : \Sigma}$$

$\boxed{P \vdash H; Ts : \Sigma; TT}$

T-PROGRAM-STATE
$$\frac{\begin{array}{c}\vdash P \qquad \vdash H : \Sigma \qquad \forall i : TT(i) = (\phi_i, \Upsilon_i; \Gamma_i) \qquad \forall i : Ts(i) = (V_i, Ls_i, e_i) \qquad \forall i : \phi_i; \Gamma_i; Ls_i \vdash H \\ \forall i : H; V_i \vdash \Upsilon_i \qquad \forall i : \Sigma \vdash V_i : \Gamma_i \qquad \forall i : Ls_i \vdash e_i \qquad \forall i : H \vdash_i Ls_i \qquad \forall i : \Sigma; \phi_i; \Upsilon_i; \Gamma_i; [] \vdash e_i : \tau_i; \Upsilon_i' \qquad \text{CompleteForest(H)} \\ \biguplus_i \text{Dom}(\phi_i/\text{null}) \qquad \biguplus_i Ls_i \qquad \mathsf{ValidUniques}(H, Ts, \Sigma, TT) \qquad \biguplus_i \{l | \text{root}(x) \in \Upsilon_i \wedge \mathsf{Dup}(V_i(x)) = l\} \qquad \forall i, j : \mathsf{SeparateTrees}(H, V_i, \Upsilon_i, V_j, \Upsilon_j)\end{array}}{P \vdash H; Ts : \Sigma; TT}$$

$\boxed{\Gamma; V \vdash Ls : L}$

T-MATCH-NO-LOCKS
$$\frac{}{\Gamma; V \vdash \emptyset : \emptyset}$$

T-MATCH-LOCK
$$\frac{\Gamma(x) = \text{final}\, \tau \qquad V(x) = l \qquad \Gamma; V \vdash Ls : L}{\Gamma; V \vdash l :: Ls : x :: L}$$

$$\mathsf{ValidUniques}(H, Ts, \Sigma, TT) \equiv \forall l \in \text{Dom}(H) : \left(\begin{array}{c} \mathsf{NumUniqueFieldsHolding}(H, \Sigma, l^\bullet) + \\ \mathsf{NumUniqueVarsHolding}(Ts, TT, l^\bullet) + \\ \mathsf{NumUniqueTypedLiterals}(Ts, TT, l^\bullet) \end{array} \right) \leq 1$$

$$\mathsf{SeparateTrees}(H, V_1, \Upsilon_1, V_2, \Upsilon_2) \equiv \left(\left(\begin{array}{c} \text{subtree}(x) \in \Upsilon_1 \wedge \\ \text{subtree}(y) \in \Upsilon_2 \end{array} \right) \Rightarrow \left(\begin{array}{c} \neg\exists r \in \Upsilon_1, t \in \Upsilon_2, f_1 \ldots f_k, g_1 \ldots g_n. \\ H|_{\text{complete}}(V_1(r).f_1 \ldots f_k) = V_2(y) \wedge \\ H|_{\text{complete}}(V_2(t).g_1 \ldots g_n) = V_1(x) \end{array} \right) \right)$$

Figure 10. Program state typing and uniqueness preservation predicates.

then there exists a $\Sigma', \phi', \Upsilon'', \Gamma'$ *such that*

C1. $\Sigma'; \phi'; \Upsilon''; \Gamma'; L \vdash E[e'] : \tau; \Upsilon'''$ *(the reduced expression type checks)*

C2. $\Upsilon' \subseteq \Upsilon'''$ *(the new final tree assertions contain at least the same disjointness information as the original final assertions.*

C3. $\text{subtree}(x) \in \Upsilon'' \Rightarrow (\text{subtree}(x) \in \Upsilon \vee \neg\exists z.\text{subtree}(z) \in \Upsilon)$ *(the reduction preserves the current subtree)*

C4. $Ls' = Ls'_E @ Ls_L$ *(the new dynamic lock set is also partitionable, to those matched in the new expression and those in L)*
C5. $Ls'_E \vdash E[e']$
C6. $\Sigma \subseteq \Sigma'$
C7. $\Gamma \subseteq \Gamma'$
C8. $\mathsf{RaceFreePath}(\Gamma, L, E[e]) \Rightarrow \mathsf{RaceFreePath}(\Gamma', L, E[e'])$ *(if the expression was a race-free path, the new expression is as well)*
C9. $\mathrm{Path}(E[e]) \wedge \mathrm{Path}(e) \wedge \mathrm{Path}(e') \Rightarrow E[e'] \in \mathsf{MustAlias}(\cdot, E[e])$ *(if the expression reduced was a path, the resulting expression is a must-alias of the original path).*
C10. $\vdash H' : \Sigma'$ *(the heap is still well-typed)*
C11. $\phi'; \Gamma'; Ls' \vdash H'$ *(ϕ' reflects the capability grants for the locks held)*
C12. $H'; V' \vdash \Upsilon''$ *(disjointness assertions are still true)*
C13. $\Sigma' \vdash V' : \Gamma'$ *(local variables are still well-typed)*
C14. $H' \vdash_i Ls'$ *(the heap respects the reduced thread's new lock set)*
C15. $\Gamma' \vdash \Upsilon''$ *(Υ'' is well-formed w.r.t. Γ')*
C16. $\mathsf{CompleteForest}(H')$ *(the still heap reflects a forest-shaped capability granting relation)*
C17. $H(l) = \langle c, F, \mathrm{Some}(j)\rangle \Rightarrow j \neq i \Rightarrow H'(l) = \langle c, F, \mathrm{Some}(j)\rangle$ *(the reduction does not release or steal another thread's lock)*
C18. $\mathsf{ValidUniques}(H', (i, V', Ls', e'), \Sigma', (\phi', \Upsilon'', \Gamma'))$ *(the thread preserves uniqueness)*
C19. $l^\bullet \in (H', V', e') \Rightarrow l^\bullet \in (H, V, e) \vee l \notin \mathrm{Dom}(H)$
C20. $\mathrm{root}(x) \in \Upsilon'' \wedge \mathrm{root}(x) \notin \Upsilon \Rightarrow (\exists l. H(l) = \langle c, F, o\rangle \wedge P \vdash \mathrm{complete}\langle\mathrm{this}\rangle\ c_f\ f \in \mathrm{Fields}(c) \wedge \mathsf{Dup}(F(f)) = V(x)) \vee V'(x) \notin \mathrm{Dom}(H)$ *(any new roots were either capability-reachable or non-existent in the previous state)*
C21. $[x||y] \in \Upsilon'' \wedge [x||y] \notin \Upsilon \Rightarrow (\exists l, l', f. H_{\mathrm{complete}}(l)(f) = l' \wedge V'(x) = l' \wedge \Upsilon; \Gamma; L \vdash r \rightarrowtail l \wedge ([r||y] \in \Upsilon \vee r = y)) \vee (\exists l, l', f. H_{\mathrm{complete}}(l)(f) = l' \wedge V'(y) = l' \wedge \Upsilon; \Gamma; L \vdash r \rightarrowtail l \wedge ([x||r] \in \Upsilon \vee r = x)) \vee V'(x) \notin \mathrm{Dom}(H) \vee V'(y) \notin \mathrm{Dom}(H)$ *(any new non-reachability assertions are from just-broken reachability relationships or newly allocated objects)*
C22. $\exists l, l', \overline{f}. H'_{\mathrm{complete}}(l)(\overline{f}) = l' \wedge (\neg H_{\mathrm{complete}}(l)(\overline{f}) = l') \Rightarrow V(y) = l' \wedge V(x) = l \wedge [x||y] \in \Upsilon$ *(any new reachability was asserted as safe in the previous state)*

Proof. By induction on the derivation of $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E[e] : \tau; \Upsilon'$ (H1).

- Case T-IF:
  - Case $E \neq [\cdot]$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{if } E'[e]\ e_t\ e_f : \tau; \Upsilon'$

    By inversion on T-IF:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e] : \tau_c; \Upsilon_c$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e_t : \tau; \Upsilon_t$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e_f : \tau; \Upsilon_f$
    - $\Upsilon' = \Upsilon_t \cap \Upsilon_f$

    By induction:
    - $\exists \Sigma', \phi', \Upsilon'', \Gamma'. \Sigma'; \phi'; \Upsilon''; \Gamma'; L \vdash E'[e'] : \tau_c; \Upsilon'_c$
    - $\Upsilon_c \subseteq \Upsilon'_c$
    - $\ldots$

    Typing of the branches is preserved by Lemma 2 with potentially larger sets of tree assertions, whose intersection must therefore be a superset of Υ', so T-IF may be re-applied.
  - Case $E = [\cdot]$: By the form of T-IF (H1), the whole if construct is the redex e:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{if } v\ e_t\ e_f : \tau; \Upsilon'$

    Two local reduction rules could apply to this construct: E-IF-TRUE and E-IF-FALSE:
    - Case E-IF-TRUE: $H, i, V, Ls, \text{if } l^{[\bullet]}\ e_t\ e_f \rightarrow H, V, Ls, e_t$. By inversion on the typing derivation, $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e_t : \tau; \Upsilon_t$ (C1), and by the definition of set intersection, $\Upsilon' \subseteq \Upsilon_t$, so let $\Upsilon''' = \Upsilon_t$ (C2). Other proof goals are straightforward to derive.
    - Case E-IF-FALSE: Similar to the E-IF-TRUE subcase.
- Case T-WVAR:
  - Case $E \neq [\cdot]$: By the form of T-WVAR, the typing derivation must be of the form:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash x := E'[e] : \tau; \Upsilon'$

    By inversion on T-WVAR:
    - $\Gamma(x) = \tau_x$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e] : \tau_x; \Upsilon'$
    - $\tau = \mathsf{Alias}(\tau_x)$

    By induction:
    - $\exists \Sigma', \phi', \Upsilon'', \Gamma'. \Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e'] : \tau_x; \Upsilon'''$
    - $\Upsilon' \subseteq \Upsilon'''$
    - $\ldots$

    Thus T-WVAR may be re-applied.
  - Case $E = [\cdot]$: By the form of T-WVAR, the whole assignment construct is the redex e:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash x := v : \tau; \Upsilon'$

    By inversion on T-WVAR and Lemma 1:
    - $\Gamma(x) = \tau_x$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \tau_x; \Upsilon$
    - $\tau = \mathsf{Alias}(\tau_x)$

    Only one local reduction rule could apply: E-WVAR, so by inversion on H2
    - $H, i, V, Ls, x := v \rightarrow H, V[x \mapsto v], Ls, \mathsf{Dup}(v)$

    By the value typing judgements, trivially any untagged duplicate of a location literal typed at a certain type τ can be typed at $\mathsf{Alias}(\tau)$, so
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \mathsf{Dup}(v) : \tau; \Upsilon$ (C1)

    Most proof goals carry over unchanged from the previous state's typing. The remaining assertions are straightforward to derive. For example, $\Sigma \vdash V' : \Gamma$ (C13) holds because the only changed portion is the mapping for x, which was replaced with null (holds trivially) or a location with the same class type and by inversion on the value typing, correct tag. Assertions about Υ (C12,15) still hold because the modified variable was not final. Uniqueness is preserved (C18,19) because if the value was a unique reference it moved from an expression literal to a local variable, and possibly some other unique reference was overwritten depending on $\Gamma(x)$.
- Case T-VAR: By the form of T-VAR,
  - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash x : \tau; \Upsilon'$

  There is no sub-context, so x is the redex expression, and steps to some $v = \mathsf{Dup}(V(x))$ by inversion on E-VAR (H2). By inversion on T-VAR and expansion of $\mathsf{ValidCap}()$:
  - $\Gamma(x) = q\ \tau_x$
  - $\tau = \mathsf{Alias}(\tau_x)$
  - $\Upsilon = \Upsilon_e$
  - $\tau = \mathrm{borrowed}\langle y\rangle\ c \Rightarrow x \in \mathsf{MustAlias}(\cdot, p) \wedge \Gamma; L \vdash p : \mathrm{complete}\langle y\rangle\ c$

  τ is either $\mathrm{partial}\ c$ or $\mathrm{borrowed}\langle y\rangle\ c$ by the definition of $\mathsf{Alias}()$. If it is a partial type, it is straightforward to derive a typing of v by either using T-NULL or inverting on $\Sigma; \vdash V : \Gamma$ (H10) and applying T-ANY-LOCATION. If τ_e is a borrowed type, then by inversion on the typing:

- $x \in \mathsf{MustAlias}(\cdot, p) \wedge \Gamma; L \vdash p : \text{complete}\langle y \rangle\ c$

Clearly, $v \in \mathsf{MustAlias}(\cdot, x)$, so $v \in \mathsf{MustAlias}(\cdot, p)$ as well. By $\phi; \Gamma; Ls \vdash H$ (H8):

- $\phi(v) = y$

making it possible to apply T-BORROWED-VALUE. In either case:

- $\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \tau; \Upsilon$ (C1)

x and v are both race-free paths (C8), and share any final aliases (C9). All other proof goals either carry over directly from the previous state or are straightforward to derive.

- Case T-FIELD:
  - Case $E \neq [\cdot]$: By the form of T-FIELD:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e].f : \tau; \Upsilon$

    By inversion on T-FIELD and unfolding $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e] : \alpha\ c; \Upsilon$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, E'[e], x)$
    - $\mathsf{RaceFreePath}(\Gamma, L, E'[e])$
    - $P \vdash \tau_f\ f \in \text{Fields}(c)$
    - $\tau = \mathsf{Alias}(\tau_f[x/\text{this}])$

    By induction:
    - $\Sigma'; \phi'; \Upsilon''; \Gamma'; L \vdash E'[e'] : \alpha\ c; \Upsilon'''$
    - $\Upsilon \subseteq \Upsilon'''$
    - $\ldots$
    - $\mathsf{RaceFreePath}(\Gamma', L, E'[e'])$
    - $E'[e'] \in \mathsf{MustAlias}(\cdot, E'[e])$

    By the aliasing result:
    - $\mathsf{LockedFinalAlias}(\Gamma', L, E'[e'], x)$

    Other proof obligations are straightforward to derive.
  - Case $E = [\cdot]$: This means the whole expression is the redex, so:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l.f : \tau; \Upsilon$ (H1)

    By inversion on T-FIELD and unfolding $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \alpha\ c; \Upsilon$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, l, x)$
    - $\mathsf{RaceFreePath}(\Gamma, L, l)$
    - $P \vdash \tau_f\ f \in \text{Fields}(c)$
    - $\tau = \mathsf{Alias}(\tau_f[x/\text{this}])$

    Only one local reduction could apply to this expression, E-FIELD:
    - $H, tid, V, Ls, l.f \rightarrow H, V, Ls, \mathsf{Dup}(H(l)(f))$ (H2)

    Let $v = \mathsf{Dup}(H(l)(f))$ be the new expression. τ is either a partial reference type or a borrowed reference type. Typing v as a partial reference of the correct class is straightforward by $\vdash H : \Sigma$ (H7) and T-ANY-LOCATION or T-NULL. If it is a borrowed type $\text{borrowed}\langle x \rangle\ c$, then by the value's origin (field types may only be partial or complete):
    - $l.f \in \mathsf{MustAlias}(\cdot, v)$
    - $\Gamma; L \vdash l.f : \text{complete}\langle x \rangle\ c$

    And by $\phi; \Gamma; Ls \vdash H$ (H8):
    - $\phi(v) = x$

    So T-BORROWED-VALUE can be applied. In either case:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \tau; \Upsilon$ (C1)

    Clearly, $v \in \mathsf{MustAlias}(\cdot, l.f)$. $l.f$ and v are both race free paths (C8) and share any final aliases (C9), proving the race-free-path and aliasing obligations. Other proof obligations are straightforward to prove.
- Case T-DVAR: By the form of T-DVAR,
  - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{dread}(x) : \tau; \Upsilon'$

  There is no sub-context, so $\text{dread}(x)$ is the redex expression, and steps to some $v = V(x)$ (H2). By inversion on T-DVAR:
  - $\Gamma(x) = \text{guardless}\ c$
  - $\tau = \text{guardless}\ c$
  - $\Upsilon' = \Upsilon$
  - $\mathsf{FinalAlias}(\Gamma, x, y)$
  - $\text{root}(y) \in \Upsilon$

  Only one local reduction rule applies to this syntax, E-DVAR:
  - $H, tid, V, Ls, \text{dread}(x) \rightarrow H, V[x \mapsto \text{null}], Ls, V(x)$

  Let $v = V(x)$ be the new expression. It is straightforward to apply T-UNIQUE-NULL or T-UNIQUE-LOC to derive
  - $\Sigma; \phi; \Gamma; \Upsilon; L \vdash v : \tau; \Upsilon$ (C1)

  by exploiting the fact that x and v were aliased in the initial state, and v will share any of x's aliases from that state (the value is appropriately tagged by inversion on $\Sigma \vdash V : \Gamma$, H10). Most other proof obligations carry over directly from the previous state. For the others:
  - Uniqueness is preserved (C18,19): a unique reference moved from a local variable to a literal.
  - $\Sigma \vdash V' : \Gamma$ (C13) is straightforward because V' is unchanged from V except for one non-final variable now mapping to null.
  - $H; V' \vdash \Upsilon$ (C12) is straightforward to rederive because no final variable or heap cell changed.
- Case T-WFIELD-COMPLETE:
  - Case $E = E'[e].f := e_2$: By inversion on T-WFIELD-COMPLETE and unfolding the definition of $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e] : \tau_p; \Upsilon$
    - $\mathsf{RaceFreePath}(\Gamma, L, E'[e])$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, E'[e], x)$
    - $P \vdash \tau_f\ f \in \text{Fields}(\text{Class}(\tau_p))$
    - $\tau' = \tau_f[x/\text{this}] = \text{complete}\langle x \rangle\ c$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e_2 : \text{guardless}\ c; \Upsilon_2$
    - $y \in \mathsf{MustAlias}(\cdot, e_2)$
    - $\text{root}(y) \in \Upsilon_2$
    - $\Upsilon_2; \Gamma; L \vdash r \rightarrowtail E'[e]$
    - $[r||y] \in \Upsilon_2$
    - $\Upsilon' = \Upsilon_2/y$
    - $\tau = \text{borrowed}\langle x \rangle\ c$

    By induction:
    - $\Sigma'; \phi'; \Upsilon''; \Gamma'; L \vdash E'[e'] : \alpha\ c; \Upsilon'''$
    - $\Upsilon \subseteq \Upsilon'''$
    - $\ldots$
    - $\mathsf{RaceFreePath}(\Gamma', L, E'[e'])$
    - $E'[e'] \in \mathsf{MustAlias}(\cdot, E'[e])$

    By the aliasing result:
    - $\mathsf{LockedFinalAlias}(\Gamma', L, E'[e'], x)$

    Other proof obligations are straightforward to derive.
  - Case $E = l.f := E'[e]$: Similar to the previous subcase.
  - Case $E = [\cdot]$: The whole field write is the redex:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l.f := v : \tau; \Upsilon'$ (H1)

    By inversion on T-WFIELD-COMPLETE and unfolding $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \tau_p; \Upsilon$
    - $\mathsf{RaceFreePath}(\Gamma, L, l)$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, l, x)$
    - $P \vdash \tau_f\ f \in \text{Fields}(\text{Class}(\tau_p))$
    - $\tau' = \tau_f[x/\text{this}] = \text{complete}\langle x \rangle\ c$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \text{guardless}\ c; \Upsilon_2$
    - $y \in \mathsf{MustAlias}(\cdot, v)$
    - $\text{root}(y) \in \Upsilon_2$
    - $\Upsilon_2; \Gamma; L \vdash r \rightarrowtail l$

- $[r||y] \in \Upsilon_2$
- $\Upsilon' = \Upsilon_2/y$
- $\tau = \text{borrowed}\langle x \rangle\ c$

Let $\phi' = \phi[\mathsf{Dup}(v) \mapsto x]$, additionally removing l_0 if initially $H(l)(f) = l_0^\bullet$ (i.e. the field write overwrote a unique reference) and removing null if no other unique field of the modified object holds null after the write. It is straightforward to apply T-BORROWED-VALUE to derive

- $\Sigma; \phi'; \Upsilon'; \Gamma; L \vdash v : \text{borrowed}\langle x \rangle\ c; \Upsilon'$ (C1)

Because only the thread expression and a single heap field were modified, most proof obligations follow almost directly from hypotheses. We must prove the others:

- (C11) $\phi'; Ls \vdash H'$: This is trivial if $v = \text{null}$. Otherwise, $v = l'^\bullet$ (by inversion on the value typing) for some $l'^\bullet$, so $H' = H[l.f \mapsto l'^\bullet]$ and f is a complete field of the modified object, and by construction of ϕ' above we removed any location whose unique reference was overwritten. Other assertions in ϕ still hold in the new heap, allowing us to apply T-VALID-CAP-ORACLE.
- (C12) $H'; V \vdash \Upsilon'$: Because of the non-aliasing constraint on tree roots from $H; V \vdash \Upsilon$ (H9), and the fact that Υ' removes any assertions violated by the field write to create H'.
- (C15) $\Gamma \vdash \Upsilon'$: Υ' is a subset of Υ, so this is trivial.
- (C16) $\mathsf{CompleteForest}(H')$: By $H; V \vdash \Upsilon$ and assumptions from inverting T-WFIELD-COMPLETE, in H there was no path of complete references from the stored value to l. So by adding a complete reference from l to the stored value (if non-null), no cycle was introduced.

Most other proof obligations are straightforward to derive, including the lemma result that this newly-created capability reachability was the result of a disjointness assertion in the previous state (C22).

- Case T-WFIELD-PARTIAL:
  - Case $E = E'[e].f := e_2$: $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e].f := e_2 : \tau; \Upsilon'$. By inversion on T-WFIELD-PARTIAL and unfolding $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E'[e] : \tau_p; \Upsilon$
    - $\mathsf{RaceFreePath}(\Gamma, L, E'[e])$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, E'[e], x)$
    - $P \vdash \tau_f\ f \in \text{Fields}(\text{Class}(\tau_p))$
    - $\tau = \tau_f = \text{partial}\ c$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash e_2 : \tau; \Upsilon'$

    The case finishes by induction and re-application of T-WFIELD-PARTIAL.
  - Case $E = l.f := E'[e]$: Similar to the previous case.
  - Case $E = [\cdot]$: The whole expression is the redex:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l.f := v : \tau; \Upsilon'$ (H1)

    Only one local evaluation rule could apply: E-WFIELD (H2):
    - $H, tid, V, Ls, l.f := v \rightarrow H[l.f \mapsto v], V, Ls, \mathsf{Dup}(v)$

    By inversion on T-WFIELD-PARTIAL, Lemma 1, and unfolding $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \tau_l; \Upsilon$
    - $\mathsf{RaceFreePath}(\Gamma, L, l)$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, l, x)$
    - $P \vdash \tau_f\ f \in \text{Fields}(\text{Class}(\tau_l))$
    - $\tau_e = \tau_f = \text{partial}\ c$
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \tau; \Upsilon$
    - $\Upsilon = \Upsilon'$

    Applying T-ANY-LOCATION or T-NULL anew after inverting, it is straightforward to rederive:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \mathsf{Dup}(v) : \tau; \Upsilon'$ (C1)

    Other proof obligations are straightforward to derive.
- Case T-DFIELD:
  - Case $E \neq [\cdot]$:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{dread}(E'[e].f) : \text{guardless}\ c; \Upsilon'$

    This case proceeds much like the inductive cases for T-WFIELD-PARTIAL and T-WFIELD-COMPLETE.
  - Case $E = [\cdot]$: The whole expression must be the redex:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{dread}(l.f) : \text{guardless}\ c; \Upsilon'$ (H1)

    Only one local evaluation rule could apply, E-DFIELD (H2):
    - $H, tid, V, Ls, \text{dread}(l.f) \rightarrow H[l.f \mapsto \text{null}], V, Ls, H(l)(f)$

    By inversion on T-DFIELD, Lemma 1 and unfolding $\mathsf{FieldAccess}()$:
    - $\Sigma; \phi; \Gamma; \Upsilon; L \vdash l : \tau_l; \Upsilon$
    - $\mathsf{RaceFreePath}(\Gamma, L, l)$
    - $\mathsf{LockedFinalAlias}(\Gamma, L, l, x)$
    - $P \vdash \tau_f\ f \in \text{Fields}(\text{Class}(\tau_l))$
    - $\tau' = \tau_f[x/\text{this}] = \text{complete}\langle x \rangle\ c$
    - $\mathsf{FinalAlias}(\Gamma, l.f, t)$
    - $t \notin \Upsilon$
    - $\Upsilon; \Gamma; L \vdash r \rightarrowtail p$
    - $\Upsilon' = \Upsilon \cup \{\text{root}(t)\} \cup \{t||z | \forall z.[r||z] \in \Upsilon'\}$

    Let $v = H(l)(f)$ be the new expression. Let ϕ' be ϕ without the mapping $v \mapsto y$ for any y aliased to l if v is a location, otherwise $\phi' = \phi$. It is straightforward to apply T-UNIQUE-NULL or T-UNIQUE-LOC to derive:
    - $\Sigma; \phi'; \Gamma; \Upsilon' \vdash v : \text{guardless}\ c; \Upsilon'$ (C1)

    Many assertions carry over directly or in obvious ways from assertions about the initial state. For the others:
    - (C11) $\phi'; \Gamma; Ls \vdash H'$: By construction ϕ' is appropriately weakened to reflect the change in the heap.
    - (C12) $H'; V \vdash \Upsilon''$: By construction of the new heap and the uniqueness assertion of the previous state, v is a root; there is no complete reference in the heap that points to it. Similarly, the added tree disjointness assertions also match the new heap. All added assertions use t, which is a root in Υ''. Any variable used in an assertion in Υ'' must have a corresponding root or subroot assertion. t's referent was not a root in the previous state, so no root variable in Υ'' aliases t. By $t \notin \Upsilon$, there there is no subtree assertion for t. Aliased tree roots are not asserted to be (transitively) disjoint from each other, because the new tree is only asserted as disjoint from the single tree it was severed from.
    - (C15) $\Gamma \vdash \Upsilon''$: By $\Gamma \vdash \Upsilon$ and construction of Υ'', Υ'' only uses final variable names.
    - (C16) $\mathsf{CompleteForest}(H')$: Because $\mathsf{CompleteForest}(H)$ and H' removes a single complete reference edge.
    - (C18,19) Uniqueness is preserved; a unique reference moves from a field to an expression literal.
    - (C20,21) The new root in the tree assertions comes from a reference that was not previously a root, and the new tree disjointness assertions refer to this new root.
- Case T-NEW: The allocation construct has no inner evaluation contexts, so this expression is the whole redex. Only one local reduction rule could apply, E-NEW (H2):
  - $H, tid, V, Ls, \text{let guardless}\ x,\ \text{final}\ y = \text{new}\ c\ \text{in}\ e_{body} \rightarrow H[l \mapsto \langle c, \mathit{NullFields}_c, \text{None} \rangle], V[x \mapsto l^\bullet, y \mapsto l], Ls, e_{body}$

By inversion on T-NEW (H1):

- $\Upsilon_{body} = \Upsilon \cup \{\text{root}(y)\} \cup \{y||z|\forall z.\text{tree}(z) \in \Upsilon\}$
- $\Sigma; \phi; \Upsilon_{body}; \Gamma[x \mapsto \text{guardless } c][y \mapsto \text{final partial } c]; L \vdash e : \tau; \Upsilon_b$
- $\Upsilon_e = \Upsilon_b / y$

Let $\Gamma' = \Gamma[x \mapsto \text{guardless } c][y \mapsto \text{final partial } c]$. Let $\Sigma' = \Sigma[l \mapsto c]$. By construction of H' and Σ', and $\vdash H : \Sigma$ (H7), clearly

- $\vdash H' : \Sigma'$ (C10)

Most obligations are straightforward to derive because they are either straightforward to extend for the small change to the heap and expression, or are assertions about unchanged state. By construction of V', Σ', and Γ', and $\Sigma \vdash V : \Gamma$ (H10):

- $\Sigma' \vdash V' ; \Gamma'$ (C13)

By construction, Υ_{body} only adds assertions to Υ using variables that were either already in Υ and are therefore final by $\Gamma \vdash \Upsilon$ (H12), or refer to y, which is final in Γ', so

- $\Gamma' \vdash \Upsilon_{body}$ (C15)

$H'; V' \vdash \Upsilon_{body}$ (C12) is straightforward to derive, as are the uniqueness obligations:

- (C18) $\mathsf{ValidUniques}(H', (i, V', Ls, e_{body}), \Sigma', (\phi, \Upsilon_{body}, \Gamma'))$.
- (C19) $l^\bullet \in (H', V', e') \Rightarrow l^\bullet \in (H, V, e) \vee l \notin \text{Dom}(H)$

Because l was not in the domain of the previous state's heap, the added assertions about non-reachability of the new allocation satisfy the implication about the new tree assertions being larger than the previous (C2). The proof obligations constraining new tree roots and disjointness assertions are also satisfied, because those additions refer to the new allocation (C20,21).

- Case T-LET:
  - Case $E \neq [\cdot]$: Proceeds by inversion on T-LET, induction, and Lemma 2 (environment strengthening).
  - Case $E = [\cdot]$: The let expression is the current redex. Only one local reduction rule applies:
    - $H, tid, V, Ls, \text{let } q\ x = v \text{ in } e_{body} \rightarrow H, V[x \mapsto v], Ls, e_{body}$ (H2)

    By inversion on T-LET (H1) and Lemma 1:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash v : \tau_1; \Upsilon$
    - $\Sigma; \phi; \Upsilon; \Gamma[x \mapsto q\ \tau_1]; L \vdash e_{body} : \tau; \Upsilon_b$ where the final modifier is present only if it was present in the original expression
    - $\Upsilon' = \Upsilon_b / x$

    We already have a typing for the new expression. Clearly $\Upsilon' \subseteq \Upsilon_b$ (C2). If the variable bound is final, then ϕ must be extended by Lemma 2 with mappings to x for any mappings to final aliases of x to prove $\phi'; \Gamma'; Ls \vdash H$ (C11). Other assertions of well-typed states are either unchanged from previous states; or are straightforward to derive by the extension of V and Γ and the typing for v from inverting on T-LET, like $\Sigma \vdash V' : \Gamma'$ (C13), $H; V' \vdash \Upsilon$ (C12), and $\Gamma' \vdash \Upsilon$ (C15). Uniqueness is clearly preserved (C18,19); the step either does not affect uniqueness, or moves a unique reference from an expression literal to a local variable.
- Case T-LOCK-FIRST:
  - Case $E \neq [\cdot]$: Proceeds by inversion on T-LOCK-FIRST, induction, and Lemma 2 (environment strengthening).
  - Case $E = [\cdot]$: The lock expression is the redex:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{lock } l\ e_{body} : \tau; \Upsilon'$ (H1)
    - $L = []$ (H1)

    Because $L = []$ and $\Gamma; V \vdash Ls_L : L$, $Ls_L = []$, and the redex contains no withlocks, so $Ls_E = []$ and therefore $Ls = []$. Thus only one local reduction could have applied, E-LOCK (H2):
    - $H, i, V, Ls, \text{lock } l\ e \rightarrow H[l \mapsto \langle c, F, \text{Some}(i)\rangle], V, l :: Ls, \text{withlock } l\ e$ where $H(l) = \langle c, F, \text{None}\rangle$

    By inversion on T-LOCK-FIRST and Lemma 1:
    - $\Sigma; \phi; \Upsilon; \Gamma; [] \vdash l : \tau_l; \Upsilon$
    - $\mathsf{RaceFreePath}(\Gamma, [], l^{[\bullet]})$
    - $\mathsf{FinalAlias}(\Gamma, l^{[\bullet]}, x)$
    - $\Upsilon_t = \mathsf{NewSubtrees}([], \Upsilon, x)$
    - $\neg\exists z.\text{subtree}(z) \in \Upsilon$
    - $\Sigma; \phi; \Upsilon \cup \Upsilon_t; \Gamma; [x] \vdash e_{body} : \tau; \Upsilon_b$
    - $\tau_e = \text{borrowed}\langle y\rangle\ c \Rightarrow y \in []$
    - $\Upsilon' = \Upsilon_b / \{z | z \in \Upsilon_t\}$

    It is straightforward to apply T-WITHLOCK with exactly those hypotheses to derive:
    - $\Sigma; \phi; \Upsilon \cup \Upsilon_t; \Gamma; [] \vdash \text{withlock } l\ e_{body} : \tau; \Upsilon'$

    Let $\phi' = \phi[l' \mapsto x | \forall l'.l'$ is stored in a unique field of $l]$. By Lemma 2, and setting $\Upsilon'' = \Upsilon \cup \Upsilon_t$:
    - $\Sigma; \phi'; \Upsilon''; \Gamma; [] \vdash \text{withlock } l\ e_{body} : \tau; \Upsilon'$ (C1)

    By construction of Υ'' and the assumption that there was no subtree assertion in Υ, we maintained well-formedness of Υ'', particularly that there is at most subtree assertion:
    - $H'; V \vdash \Upsilon''$ (C12)

    Because by assumption we know there was no subtree assertion in Υ, we satisfy the constraint on new subtree assertions:
    - $\text{subtree}(x) \in \Upsilon'' \Rightarrow (\text{subtree}(x) \in \Upsilon \vee \neg\exists z.\text{subtree}(z) \in \Upsilon)$ (C3)

    By CS-NO-MORE:
    - $[l] \vdash \text{withlock } l\ e_{body}$

    By construction of the new heap:
    - $H(l) = \langle c, F, \text{Some}(j)\rangle \Rightarrow j \neq i \Rightarrow H'(l) = \langle c, F, \text{Some}(j)\rangle$ (C17)

    By $H \vdash_i Ls$ (H11, where $Ls = []$) and construction of the new heap H':
    - $H' \vdash_i [l]$ (C14)

    By $\phi; \Gamma; Ls \vdash H$ (H8) and construction of ϕ' and H':
    - $\phi'; \Gamma; l :: Ls \vdash H'$ (C11)

    By $Ls_E \vdash E[e]$ (H5), the transformation step taken, and CS-LOCK-HELD:
    - $l :: Ls_E \vdash E[e']$ (C5)
    - $Ls'_E = l :: Ls_E$
    - $Ls' = Ls'_E @ Ls_L$ (C4)

    Other proof obligations are straightforward to derive.
- Case T-LOCK-N:
  - Case $E \neq [\cdot]$: Proceeds by inversion on T-LOCK-N, induction, and Lemma 2 (environment strengthening).
  - Case $E = [\cdot]$: This case is almost identical to the T-LOCK-FIRST base case, except for a few additional hypotheses from inverting T-LOCK-N (used in the deadlock freedom proof, but not directly relevant to type preservation), and the fact that either E-LOCK or E-RECLOCK could have performed the local reduction (H2) because the static lock set L was not constrained to be empty.
- Case T-WITHLOCK:
  - Case $E \neq [\cdot]$: By the structure of withlock expressions:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{withlock } l\ E'[e] : \tau; \Upsilon'$

    By inversion on T-WITHLOCK and Lemma 1:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \tau_l; \Upsilon$
    - $\mathsf{FinalAlias}(\Gamma, l, x)$
    - $\Upsilon_t = \mathsf{NewSubtrees}(L, \Upsilon, x)$
    - $\forall z.\text{subtree}(z) \in \Upsilon_t \Rightarrow \text{subtree}(z) \in \Upsilon$

– $\Sigma; \phi; \Upsilon; \Gamma; x :: L \vdash E'[e] : \tau; \Upsilon_b$
– $\tau = \text{borrowed}\langle y \rangle\ c \Rightarrow y \in L$
– $\Upsilon' = \Upsilon_b / \{z | z \in \Upsilon_t\}$

By the hypotheses above, $\Gamma; V \vdash l :: Ls_L : x :: L$, so by induction:

– $\Sigma'; \phi'; \Upsilon''; \Gamma'; x :: L \vdash E'[e'] : \tau; \Upsilon'''$
– $\Upsilon' \subseteq \Upsilon'''$
– $\text{subtree}(x) \in \Upsilon'' \Rightarrow \text{subtree}(x) \in \Upsilon \vee \neg\exists z.\text{subtree}(z) \in \Upsilon$
– ...
– $Ls' = Ls''_E @ (l :: Ls_L)$
– $Ls''_E \vdash E'[e']$

By CS-LOCK-HELD and letting $Ls'_E = Ls''_E @ [l]$:

– $Ls'_E \vdash E[e']$
– $Ls' = Ls'_E @ Ls_L$

Other proof obligations follow directly from induction results, using Lemma 2 and the subtree preservation (C3) to apply T-WITHLOCK with the inductive results to derive a thread typing (C1).

- Case $E = [\cdot]$: The whole withlock expression is the current redex. Two local reduction rules could have applied: E-UNLOCK and E-RECUNLOCK (H2). The two are very similar, so we describe only the former case in detail.
  – $H, i, V, l :: Ls', \text{withlock}\, l\, v \to [l \mapsto \langle c, F, \text{None}\rangle], V, Ls', v$ where $l \notin Ls'$ and $H(l) = \langle c, F, \text{Some}(i)\rangle$.

  Clearly $Ls_E = [l]$, $Ls'_E = [\,]$, $Ls'_E \vdash E[v]$ and $Ls' = Ls'_E @ Ls_L$. By inversion on T-WITHLOCK (H1) and Lemma 1:
  – $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l : \tau_l; \Upsilon$
  – $\mathsf{FinalAlias}(\Gamma, l, x)$
  – $\Upsilon_t = \mathsf{NewSubtrees}(L, \Upsilon, x)$
  – $\forall z.\text{subtree}(z) \in \Upsilon_t \Rightarrow \text{subtree}(z) \in \Upsilon$
  – $\Sigma; \phi; \Upsilon; \Gamma; x :: L \vdash E'[e] : \tau; \Upsilon_b$
  – $\tau = \text{borrowed}\langle y \rangle\ c \Rightarrow y \in L$
  – $\Upsilon \cup \Upsilon_t = \Upsilon_b$
  – $\Upsilon' = \Upsilon_b / \{z | z \in \Upsilon_t\}$

  Let ϕ' be ϕ less the mappings to x and its final aliases (since this is a non-recursive lock release, no alias of x will be in L by H4, $\Gamma; V \vdash Ls : L$). For any typing of v, it is possible to reconstruct it in a context with ϕ' and L rather than $x :: L$, by value typing and the fact that if τ_e was a borrowed type guarded by x (or its alias), then $x \in L$ (or its alias is) and then by construction $\phi'(v) = x$, so letting $\Upsilon'' = \Upsilon'$:
  – $\Sigma; \phi'; \Upsilon''; \Gamma; L \vdash v : \tau_e; \Upsilon$ (C1)

  Many proof obligations follow directly from hypotheses about the previous state because the heap structure and local variables are unchanged. For others:
  – By construction (marking that the lock is released), $H' \vdash_i Ls'$ (C14), and other threads' lock ownership as recorded in the heap is preserved (C17).
  – By construction of ϕ', $\phi'; \Gamma; Ls' \vdash H'$ (C11).
  – Uniqueness is clearly preserved (C18,19).
- Case T-SUB-TYPE: By induction and fresh application of T-SUB-TYPE.
- Case Value Typing: For T-ANY-LOCATION, T-UNIQUE-NULL, T-UNIQUE-LOC, T-BORROWED-VALUE, and T-NULL. None of these rules type expressions with subcontexts, so the evaluation context must be $[\cdot]$ and the redex is the value itself. But there is no evaluation rule that steps from a value, which violates the assumption $H, i, V, Ls, e \to H', V', Ls', e'$ (H2), so these cases hold vacuously.
- Case T-SPAWN: Similar to the previous set of cases, there is no local evaluation step of the form $H, i, V, Ls, e \to H', V', Ls', e'$ (H2) when e is a spawn expression, so this case holds vacuously.

□

Lemma 4 (Type Preservation)**.** *If* $P \vdash H; Ts : \Sigma; TT$ *and* $H; Ts \to H'; Ts'$ *then*

- *If a lock expression was reduced and was typed with* T-LOCK-N*, its lock was typed as a borrowed reference with the same lock group as its complete reference.*
- $\exists \Sigma', TT'.\Sigma \subseteq \Sigma'$*,* TT *differs from* TT' *only for the threads reduced.*

Proof. By induction on the derivation of $H; Ts \to H'; Ts'$.

- Case E-THREAD: By inversion on E-THREAD:
  - $H, (i, V, Ls, E[e]) \to H', (i, V', Ls', E[e'])$

  By inversion on E-CONTEXT:
  - $H, i, V, Ls, e \to H', V', Ls', e'$

  By inversion on T-PROGRAM-STATE, using $\phi = \phi_i$, $\Upsilon = \Upsilon_i$, etc. and eliding the results quantifying many assertions over all threads:
  - $\vdash P$ (the program is well-typed)
  - $\vdash H : \Sigma$ (the heap is well-typed)
  - $\phi; \Gamma; Ls \vdash H$ (ϕ reflects only the capability grants for the locks held)
  - $H; V \vdash \Upsilon$ (the heap and local variables are accurately modeled by Υ)
  - $\Sigma \vdash V : \Gamma$ (local variables are typed correctly)
  - $Ls \vdash E[e]$ (the dynamic lock set matches the unreduced withlock expressions)
  - $H \vdash_i Ls$ (the heap respects the reduced thread's lock set)
  - $\Gamma \vdash \Upsilon$ (Υ is well-formed w.r.t. Γ; it references only final variables)
  - $\Sigma; \phi; \Upsilon; \Gamma; [] \vdash E[e] : \tau; \Upsilon'$ (the thread's expression type checks)
  - $\mathsf{CompleteForest}(H)$ (the heap reflects a forest-shaped capability granting relation)
  - $\biguplus_i \text{Dom}(\phi_i)$ (no lock's guard is known to multiple threads)
  - $\biguplus_i Ls_i$ (the lock sets of all threads are disjoint)
  - $\mathsf{ValidUniques}(H, Ts, \Sigma, TT)$ (each object has no more than 1 unique reference)
  - $\biguplus_i \{l | \text{root}(x) \in \Upsilon_i \wedge \mathsf{Dup}(V_i(x)) = l\}$ (no more than one thread may assert a lock is the root of a capability-granting tree)
  - $\forall i, j : \mathsf{SeparateTrees}(H, V_i, \Upsilon_i, V_j, \Upsilon_j)$ (there is an overlap of tree disjointness assertions between threads that would tie two threads' tree assertions together too closely; that overlap is not possible to create, but this assertion explicitly states that it has not been produced)

  By Lemma 3, there exists a $\Sigma', \phi', \Upsilon'', \Gamma', \Upsilon'''$ such that
  - $\Sigma'; \phi'; \Upsilon''; \Gamma'; [\,] \vdash E'[e'] : \tau; \Upsilon'''$
  - $\Upsilon' \subseteq \Upsilon'''$
  - $\text{subtree}(x) \in \Upsilon'' \Rightarrow (\text{subtree}(x) \in \Upsilon \vee \neg\exists z.\text{subtree}(z) \in \Upsilon)$ (the reduction preserves the current subtree)
  - $\vdash H' : \Sigma'$
  - $\phi'; \Gamma'; Ls' \vdash H'$
  - $H'; V' \vdash \Upsilon''$
  - $\Sigma' \vdash V' : \Gamma'$
  - $Ls' \vdash E'[e']$

- $H' \vdash_i Ls'$
- $\Gamma' \vdash \Upsilon''$
- $\mathsf{CompleteForest}(H')$
- $H(l) = \langle c, F, \mathrm{Some}(j)\rangle \Rightarrow j \neq i \Rightarrow H'(l) = \langle c, F, \mathrm{Some}(j)\rangle$
- $\mathsf{ValidUniques}(H', (i, V', Ls', e'), \Sigma', (\phi', \Upsilon'', \Gamma'))$
- $l^\bullet \in (H', V', e') \Rightarrow l^\bullet \in (H, V, e) \vee l \notin \mathrm{Dom}(H)$
- $\mathrm{root}(x) \in \Upsilon'' \wedge \mathrm{root}(x) \notin \Upsilon \Rightarrow (\exists l.H(l) = \langle c, F, o\rangle \wedge P \vdash \mathrm{complete}\langle\mathrm{this}\rangle\ c_f\ f \in \mathrm{Fields}(c) \wedge \mathsf{Dup}(F(f)) = V(x)) \vee V'(x) \notin \mathrm{Dom}(H)$
- $[x||y] \in \Upsilon'' \wedge [x||y] \notin \Upsilon \Rightarrow (\exists l, l', f.H_{\mathrm{complete}}(l)(f) = l' \wedge V'(x) = l' \wedge \Upsilon; \Gamma; L \vdash r \rightarrowtail l \wedge ([r||y] \in \Upsilon \vee r = y)) \vee (\exists l, l', f.H_{\mathrm{complete}}(l)(f) = l' \wedge V'(y) = l' \wedge \Upsilon; \Gamma; L \vdash r \rightarrowtail l \wedge ([x||r] \in \Upsilon \vee r = x)) \vee V'(x) \notin \mathrm{Dom}(H) \vee V'(y) \notin \mathrm{Dom}(H)$ (any new non-reachability assertions are from just-broken reachability relationships or newly allocated objects)
- $\exists l, l', \overline{f}.H'_{\mathrm{complete}}(l)(\overline{f}) = l' \wedge (\neg H_{\mathrm{complete}}(l)(\overline{f}) = l') \Rightarrow V(y) = l' \wedge V(x) = l \wedge [x||y] \in \Upsilon$ (any new reachability was asserted as safe in the previous state)

These results are sufficient to update most typing assertions for the reduced thread, and preservation for other threads' typings follows from Lemma 2, with a few exceptions:

- $\biguplus_i Ls'_i$: If $Ls'_i \subseteq Ls_i$ this is trivial. Otherwise, any lock in Ls'_i but not in Ls_i must not have been in the dynamic lock set of any other thread of the previous state, by $\forall j : Ls_j \vdash_j H$ and the lemma result that other threads' locks are preserved. So the lock sets must still be disjoint.
- $\biguplus_i \mathrm{Dom}(\phi'_i/\mathrm{null})$: Because by $\phi'; \Gamma'; Ls' \vdash H'$ and the lock set disjointness just proven, any additions to ϕ'_i's non-null domain would be a location not in any other thread's ϕ_j.
- $\mathsf{ValidUniques}(H', Ts', \Sigma', TT')$ Follows from the individual thread uniqueness preservation results.
- $\biguplus_i \{l | \mathrm{root}(x) \in \Upsilon'_i \wedge \mathsf{Dup}(V'(x)) = l\}$: From $\forall j : H; V_j \vdash \Upsilon_j$ on the previous state and the restrictions on the growth of Υ to Υ'' from the lemma results.
- $\forall i, j : \mathsf{SeparateTrees}(H', V'_i, \Upsilon'_i, V'_j, \Upsilon'_j)$: If no complete fields are overwritten, this property is clearly preserved. If some complete field was mutated, we perform a case analysis based on $H, i, V, Ls, e \rightarrow H', V', Ls', e'$. By inspection of that set of rules, there are only two applicable rules: E-WFIELD-COMPLETE and E-DFIELD (no other rules modify fields that affect the capability-granting relation).
  - Case E-WFIELD-COMPLETE:
    To prove $\forall i, j\mathsf{SeparateTrees}(H', V'_i, \Upsilon'_i, V'_j, \Upsilon'_j)$, we prove by contradiction.
    Assume a (sub)tree t rooted at the stored value in the reduced thread capability-reaches a subtree x in the other thread, and a (sub)tree r in the other thread capability-reaches the root that (transitively) grants the capability for l in the reduced thread; in this case only would the complete-field-write violate the implication in the tree separation invariant. But by the fact that it is in an assertion and reachable, that root-of lookup result must be a subtree in the reduced thread, which could only occur if the implication was already violated in the previous state, which was not the case.
    To show that no unreachability assertion in another thread was invalidated, consider the possible relationships between the values referenced by $[a||b] \in \Upsilon_j$ for another thread j, and the assertion $[z||y] \in \Upsilon$ where z is the root transitively granting the capability for l. Most cases are trivially safe, or hold vacuously because they violate the assumption that one of the variable in a tree disjointness assertion must be a root. The only other case is one where a is capability-reachable from z, and y is capability-reachable from b. By $H; V \vdash \Upsilon$ for each thread, a and y must be subtrees. But this would violate the tree separation invariant of the previous state, which we know held, so this case holds vacuously. Thus we may apply T-PROGRAM-STATE to the new state using the previous state's typings for other threads.
  - Case E-DFIELD: $\forall i, j : \mathsf{SeparateTrees}(H, V_i, \Upsilon_i, V_j, \Upsilon_j)$ is clearly preserved.
    Other threads' non-reachability assertions remain valid (the transition did not make anything capability-reachable where it wasn't before). Then apply T-PROGRAM-STATE.

- Case E-SPAWN:
  - $H, \{(i, V, Ls, E[\mathrm{spawn}\ x\ e])\} \cup Ts_0 \rightarrow H, \{(i, V', Ls, E[\mathrm{null}]), (j, V_{new}, [], e)\} \cup Ts_0$

  By inversion on E-SPAWN:
  - j is a fresh thread ID
  - $V' = V[x \mapsto \mathrm{null}]$
  - $V_{new} = \{x \mapsto V(x)\}$

  By inversion on T-PROGRAM-STATE:
  - $\ldots$
  - $\Sigma; \phi_i; \Upsilon_i; \Gamma_i; [\,] \vdash E[\mathrm{spawn}\ x\ e] : \tau_i; \Upsilon'_i$
  - $\ldots$

  By induction on the structure of E, using T-NULL and inverting on T-SPAWN in the base case:
  - $\Gamma_i(x) = \alpha\ c$
  - $\Sigma; \emptyset; \emptyset; \{x \mapsto \text{final partial } c\}; [\,] \vdash e : \tau; \Upsilon_e$
  - $\Sigma; \phi_i; \Upsilon_i; \Gamma_i; [\,] \vdash E[\mathrm{null}] : \tau_i; \Upsilon'_i$

  Deriving the necessary hypotheses to apply T-PROGRAM-STATE to the new state is straightforward.

□

B. Source Typing Implies Runtime Typing

The proof that any program that type checks under the source type system also type checks under the runtime type system is fairly straightforward. The proof proceeds by induction on the derivation under the source type system, and relies on two main facts:

- The additional runtime contexts (Σ and ϕ) are unused in the runtime typing judgements for expressions that are also in the source language.
- Any results from a sound static the must-alias analysis on the source must also be provable by the runtime type system's oracle must-alias analysis, and the rules never negate a must-alias result.

C. Typing Implies Deadlock Freedom

The argument for how our lock capability type system prevents deadlock is not as obvious as how lock levels prevent deadlock, because the capability-granting relation in our type system can change over time, including among held locks. If it were not possible to change the capability-granting relation, deadlock freedom would follow almost directly from the fact that the capability-granting relation is acyclic. With changes to that relation, however, there is in some sense a view across time of dynamically executable lock acquisition orders, that ignores lock reordering and will remain free of cycles between locks held by different threads because the granting relation remains acyclic and locks are always acquired in accordance with the capability-granting relation.

Our strategy for proving deadlock freedom is to define an extended program semantics that explicitly models the dynamically executed capability use / lock acquisition orders, and to prove the absence of certain types of paths in this transition system as a preserved property, where those certain paths represent deadlock scenarios. It would be possible to incorporate this into the standard type preservation proof from Appendix A, but we have separated the proofs to simplify presentation.

We call the model of how capabilities are used across time the *capability-use graph*. Its vertices are held locks (as heap locations). Its edges are capability uses: if a program reduces a lock expression that typed an acquisition of location l based on the fact that x granted the capability for l, the graph picks up an edge from x's referent (say l_x) to l, $l_x \rightarrow l$. When the program releases a lock, the incoming edge to that lock is removed from the capability-use graph for that thread. If a thread reduces to a state where its next reduction would acquire another lock, we say there is an *intended edge* from the location granting the capability for that lock to that next-to-acquire lock (there is no intended edge, or incoming edge for the first lock acquired). The union of all threads' capability-use graphs (the *joint capability-use graph* and intended edges will contain no paths between locks held by the same thread that *crosses multiple threads' edges*. Those paths represent the deadlock scenarios because such a path is either:

- A dependency cycle among threads following the capability-granting relation (which contradicts the fact that the capability-granting relation remains acyclic), or
- A path from a lock held by one thread, through one or more locks held by other threads, and back to a lock held by the original thread. This would contradict the "orphaned lock" check that disallows the original thread from acquiring locks if some other thread might hold a lock that (transitively) grants the capability to acquire one of the original thread's orphaned locks.

Figure 11 gives formal definitions for the extended semantics, and properties used in the deadlock freedom proof. Because there is so much new technical machinery for the deadlock freedom proof, we first provide a sketch of a deadlock freedom proof for a system without reordering, but still using capability-use graphs for the proof. Then we outline what must change to extend the proof to handle changes in the capability-granting relation.

Proof Sketch — Without Reordering By induction on evaluation steps. By inverting on the global step, we find some local reduction that occurred in a particular thread. Those local reductions fall into one of several categories:

- **Uninteresting Reductions**: These are rules like conditional evaluation, allocation, or binding, which do not produce values or affect the set of locks held, and can be roughly ignored (really, we apply G-MATCH-REDUCTION to carry the capability-use graphs through to the next evaluation step).
- **Forbidden Reductions**: Destructive reads and writing complete references are banned for this sketch, as we consider only executions without reordering.
- **Reductions to Values**: Borrowing reads, and storing partial or borrowed values can produce values, which may be inside a lock acquisition's inner context. These reductions *may* induce intended edges in the capability-use graph for the reduced thread, when the thread reduces to some $E[\text{lock } l\ e]$. If the thread holds no locks yet, there is no intended edge (there is no source for such an edge), and no new bad paths are introduced. If the thread does hold locks, an edge showing that the thread depends on acquiring the lock l is added to the joint capability-use graph. This is trivially safe if the lock is unowned or owned by the reduced thread. If it is owned by another thread, it is still safe; this edge will correspond to some edge in the (tree-shaped) capability-granting relation, so it must be the first-lock acquired by the other thread, which then cannot transitively be blocked on any thread blocked on the reduced thread. These cases are the heart of the proof.
- **Lock Acquisition**: Reductions that acquire locks essentially just move what were previously intended edges in the joint graph to being executed edges in the thread's individual capability-use graph. Because the lock was necessarily unacquired previously, there are no outgoing edges *from* the acquired lock, so no new paths were created in the joint graph. Then because the joint graph was safe before the reduction, it is free of bad paths afterwards as well.
- **Lock Releases**: These simply remove edges from the capability-use graphs, and are trivially safe.

Adding Reordering Handling reordering primarily reduces what can be directly deduced from the capability-granting relation, because a single thread's locks may be unrelated in the capability-granting relation. To bridge such reachability gaps, we track an additional partial order of which threads *effectively* (transitively) grant the capability to acquire the *first-acquired* lock of other threads. The definition follows the natural intuition for this property; thread A effectively grants the capability to thread B's first-acquired lock if:

- There is a direct path in the capability-granting relation from some lock thread A holds to B's first lock, or
- There is a path through the capability-granting relation from the referent of a groupless reference in A's local variables or expression literal to B's first lock.

This notion of *effective granting* is used in the evaluation cases that produce values to augment information about the capability-granting relation itself when proving the absence of an incoming intended edge. Because the edges of the joint capability-use graph represent *previously executed uses* of capabilities, reordering generally leaves paths from the first lock acquired to locks held. The only exception is when reordering allows a thread to release the lock directly granting the capability to acquire another lock it still holds, in which case a gap would exist, but the type system prevents such a thread from acquiring new locks, and therefore blocking, until it releases enough locks to make its local cap-use graph contiguous.

Lemma 5 (Lock-in-Hole Typing)**.** *If*

- $\Sigma; \phi; \Upsilon; \Gamma; L \vdash E[\text{lock } l\ e] : \tau; \Upsilon'$
- $Ls = Ls_E @ Ls_L$
- $Ls_E \vdash E[\text{lock } l\ e]$
- $\Gamma; V \vdash Ls_L : L$

then

- $\Sigma; \phi; \Upsilon_s; \Gamma; L' @ L \vdash \text{lock } l\ e : \tau'; \Upsilon''$
- $L' @ L = [] \vee (\phi(l) = x \wedge V(x) = l_x \wedge x \in L' @ L \wedge (\forall z \in L' @ L : L' @ L = L''' @ [z] \vee \exists p'.\mathsf{RaceFreePath}(\Gamma, L' @ L, p') \wedge \mathsf{FinalAlias}(\Gamma, p', z) \wedge \Gamma; L' @ L \vdash p' : \text{complete}\langle a \rangle\ c))$
- $\Gamma; V \vdash Ls : L' @ L$

This lemma is used when reasoning about the edges in the joint capability-use graph that come from pending lock acquisitions (or blocking); it is essentially used to retrieve the guarding lock of a lock about to be acquired, which determines the head of the directed edge.

Capability-Use Graph	κ	::=	(V, E)
Cap-Use Graph Vertices	V	:	Location set
Cap-Use Graph Edges	E	:	Location $\rightarrow$ Location stack
Thread Graph Map	G	:	$\text{ThreadID} \rightarrow \kappa$
Joint Lock Ordering Graph	J	:	Edge-Labeled Graph

$$\textsc{G-Match-No-Locks}\quad \frac{\forall i : \mathsf{Vertices}(G(i)) = \emptyset \qquad \forall i : \mathsf{Edges}(G(i)) = [] \qquad \forall i : Ts(i) = (V_i, Ls_i, e_i) \qquad \forall i : Ls_i = []}{G \equiv Ts}$$

$$\textsc{G-Match-Reduction}\quad \frac{G_0 \equiv Ts_0 \qquad H, Ts_0 \rightarrow H', Ts \qquad \forall i : \mathsf{Vertices}(G(i)) = Ls_i \qquad G = \mathsf{StepGraph}(G_0, Ts_0, Ts)}{G \equiv Ts}$$

$$\mathsf{StepGraph}(G, Ts, Ts') = \begin{cases} G & \text{if } \forall i : Ls_i = Ls'_i \\ G & \text{if } |Ts| < |Ts'| \\ G[i \mapsto (\{l\}, [])] & \text{if } Ls_i = [] \wedge Ls'_i = [l] \\ G[i \mapsto (\mathsf{Vertices}(G(i)) \cup \{l\}, [l_0 \mapsto l] :: \mathsf{Edges}(G(i)))] & \text{if } Ls_i \neq [] \wedge l :: Ls_i = Ls'_i \wedge \\ & \quad l_0 = V_i(\phi_i(l)) \\ G[i \mapsto (\{l_v | l_v \in \mathsf{Tail}(\mathsf{Edges}(G(i)))\}, \mathsf{Tail}(\mathsf{Edges}(G(i))))] & \text{if } Ls_i = l :: Ls'_i \wedge |Ls'_i| > 1 \\ G[i \mapsto (\{l_0\}, [])] & \text{if } Ls_i = l :: Ls'_i \wedge Ls'_i = [l_0] \\ G[i \mapsto (\emptyset, [])] & \text{if } Ls_i = l :: Ls'_i \wedge Ls'_i = [] \end{cases}$$

$$\mathsf{Join}(G, TT, Ts) = \left(\bigcup_i \{l_1 \overset{i}{\mapsto} l_2 | l_1 \mapsto l_2 \in \mathsf{Edges}(G(i))\}\right) \cup \{l_0 \overset{i}{\mapsto} l | Ls_i \neq [] \wedge e_i = E_i[\text{lock } l\ e_b] \wedge l_0 = V_i(\phi_i(l))\}$$

$$\mathsf{SafeCapUse}(J, G) = \forall i. \neg\exists a, b \in \mathsf{Vertices}(G(i)). \exists l \overset{j}{\mapsto} l' \in (a \mapsto \ldots \mapsto b \subseteq J). j \neq i$$

$$\mathsf{ThreadOrder}(H, Ts, TT, i, j) \equiv Ls_j = Ls'_j @ [l_j] \wedge \left(\begin{array}{c} (\exists l_i \in Ls_i. \mathsf{CapReach}(H, l_i, l_j)) \vee \\ (\exists x_i. \Gamma_i = \text{guardless } c \wedge \mathsf{CapReach}(H, V_i(x_i), l_j)) \vee \\ (\exists l_i. l_i^\bullet \in e_i \wedge \mathsf{CapReach}(H, l_i, l_j)) \end{array} \right)$$

$$\mathsf{PartialThreadOrder}(H, Ts, TT) \equiv\neq \exists t_0 \ldots t_{n-1}. (\forall i \in (0..n-1) \mathsf{ThreadOrder}(H, Ts, TT, t_i, t_{i+1 \bmod n}))$$

Figure 11. Formal definitions for capability-use graphs, extended program states, and extended state properties.

Proof. By induction on the structure of the typing derivation. Most cases are entirely straightforward; we show only the non-trivial cases.

- Case T-WITHLOCK:
  - Case $E \neq [\cdot]$: In this case:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash \text{withlock } l_1\ E'[\text{lock } l\ e] : \tau; \Upsilon'$

    By inversion on T-WITHLOCK and Lemma 1:
    - $\Sigma; \phi; \Upsilon; \Gamma; L \vdash l_1 : \tau_1; \Upsilon$
    - $\mathsf{FinalAlias}(\Gamma, l, x)$
    - $\Upsilon_t = \mathsf{NewSubtrees}(L, \Upsilon, x)$
    - $\Sigma; \phi; \Upsilon \cup \Upsilon_t; \Gamma; x :: L \vdash E'[\text{lock } l\ e] : \tau; \Upsilon_b$
    - $\tau = \text{borrowed}\langle y \rangle\ c \Rightarrow y \in L$
    - $\Upsilon' = \Upsilon_b / \Upsilon_t$

    By T-MATCH-LOCK and the aliasing result from inversion:
    - $\Gamma; V \vdash l :: Ls_L : x :: L$

    By assumption and inversion on CS-LOCK-HELD:
    - $Ls_E = Ls'_E @ [l]$
    - $Ls'_E \vdash E'[\text{lock } l\ e]$

    By induction, using the typing of $E'[\text{lock } l\ e]$:
    - $\Sigma; \phi; \Upsilon_s; \Gamma; L''@(x :: L) \vdash \text{lock } l\ e : \tau'; \Upsilon''$
    - $L''@(x :: L) = [] \vee (\phi(l) = w \wedge V(w) = l_w \wedge w \in L'@L \wedge (\forall z \in L'@L : L'@L = L'''@[z] \vee \exists p'. \mathsf{RaceFreePath}(\Gamma, L'@L, p') \wedge \mathsf{FinalAlias}(\Gamma, p', z) \wedge \Gamma; L'@L \vdash p' : \text{complete}\langle a \rangle\ c))$
    - $\Gamma; V \vdash Ls : L''@(x :: L)$

    Let $L' = L''@[x]$.
  - Case $E = [\cdot]$: In this case the body of the withlock is a value, so this case holds vacuously.
- Case T-LOCK-FIRST:
  - Case $E \neq [\cdot]$: $E = \text{lock } E'[\text{lock } l\ e]\ e_{body}$. Straightforward inversion on the typing, plus induction.
  - Case $E = [\cdot]$: $E = [\text{lock } l\ e]$. Let $L' = []$, and by inversion on T-LOCK-FIRST, $L = []$, and therefore $L'@L = []$. Other goals follow from the inductive hypotheses.
- Case T-LOCK-N:
  - Case $E \neq [\cdot]$: $E = \text{lock } E'[\text{lock } l\ e]\ e_{body}$. Straightforward inversion on the typing, plus induction.
  - Case $E = [\cdot]$: $E = [\text{lock } l\ e]$. Let $L' = []$. By inversion on T-LOCK-N, further inversion on the typing of the lock location l via T-BORROWED-VALUE, and the induction hypotheses.

□

Lemma 6 (Capability-Use Graph Preservation)**.** *If*

- $P \vdash H; Ts : \Sigma; TT$
- $H; Ts \rightarrow H'; Ts'$
- $G \equiv Ts$
- $J = \mathsf{Join}(G, TT, Ts)$
- $\mathsf{SafeCapUse}(J, G)$
- $\mathsf{PartialThreadOrder}(H, Ts)$

then there exists a Σ', TT', G' *such that*

- $P \vdash H'; Ts' : \Sigma'; TT'$
- $G' \equiv Ts'$
- $J' = \mathsf{Join}(G', TT', Ts')$
- $\mathsf{SafeCapUse}(J', G')$
- $\mathsf{PartialThreadOrder}(H', Ts')$

Proof. By induction on $H; Ts \rightarrow H'; Ts'$:

- Case E-THREAD: By inversion on E-THREAD:

- $H, (i, V, Ls, E[e]) \to H', (i, V', Ls', E[e'])$

By inversion on E-CONTEXT:

- $H, i, V, Ls, e \to H', V', Ls', e'$

By induction on $H, i, V, Ls, e \to H', V', Ls', e'$:

- Case E-IF-TRUE, E-IF-FALSE, E-NEW, E-LET: These cases do not affect lock sets or the capability-granting relation in any nontrivial way, and because they do not reduce to values (since we use a local store rather than substitution for variables) the new thread state will not be a lock statement in an evaluation context's hole, and therefore cannot add any intended edges to the joint capability graph. The cases proceed in a straightforward manner, simply re-using hypotheses with the fact that $G' = G$ and $J' = J$ to re-apply G-MATCH-REDUCTION.
- Case E-VAR: This case is only interesting for its demonstration of dealing with the possible addition of an intended edge. The reasoning for intended edges is the same across all cases that may do so, so this case is presented in detail to show that reasoning in detail without the distraction of other invariants changing. Because the reasoning is identical in all other cases, we elide the handling of new intended edges in all cases following this one.

  By Lemma 4, there exists Σ', TT' such that

  – $P \vdash H'; Ts' : \Sigma'; TT'$

  $Ls = Ls'$, so $\mathsf{StepGraph}(G, Ts, Ts') = G$, and setting $G' = G$ it is straightforward to apply G-MATCH-REDUCTION to conclude:

  – $G' \equiv Ts'$

  Because $G' = G$ and only one thread's expression was changed, the only possible change from J to $J' = \mathsf{Join}(G', TT', Ts')$ is if $e_i' = E'[\text{lock } l\ e_b]$. In this case, by inversion on the new program state typing and Lemma 5:

  – $\Sigma'; \phi_i'; \Upsilon_{i,s}'; \Gamma_i'; L \vdash \text{lock } l\ e_b : \tau'; \Upsilon_i''$
  – $L = [] \vee (\phi_i'(l) = x \wedge V_i'(x) = l_x \wedge x \in L \wedge (\forall z \in L : L = L'@[z] \vee \exists p'.\mathsf{RaceFreePath}(\Gamma_i', L, p') \wedge \mathsf{FinalAlias}(\Gamma_i', p', z) \wedge \Gamma_i'; L \vdash p' : \text{complete}\langle a \rangle\ c))$
  – $\Gamma_i'; V_i' \vdash Ls_i' : L$

  If $L = []$, then by the last lemma result $Ls_i' = []$, and no intended edge will be added to J because $\mathsf{GuardOf}$ is undefined. Otherwise, by case analysis on the owner of l's lock:

  – $H'(l) = \langle c, F, \text{None} \rangle$. The target lock is unowned. By inversion on $G \equiv Ts$ we know that the vertices for each thread's capability use graph is the same as the thread's set of held locks. By inversion on the new program state typing we know that those are accurately reflected in the heap's lock owner fields. Adding the tentative edge $l_x \overset{i}{\mapsto} l$ to J will preserve the $\mathsf{SafeCapUse}$ assertion because no thread holds the lock on l.
  – $H'(l) = \langle c, F, \text{Some}(i) \rangle$. This leads to a recursive acquisition. The tentative edge $l_x \mapsto l$ introduced cannot introduce a multi-thread path in J' between locks held by the same thread unless there already was one in J, which by the inductive hypothesis is not the case.
  – $H'(l) = \langle c, F, \text{Some}(j) \rangle$ where $j \neq i$. This acquisition attempt will block until thread j releases lock l. The only way that adding $l_x \overset{i}{\mapsto} l$ to the joint graph would create a multi-thread cycle if if there was already a path in J from some lock held by thread j to some lock held by thread i. This could only happen through two types of paths:
    - The path ends with a tentative edge incoming to the first lock acquired by thread i. Because by the Lemma 5 results $\mathsf{ThreadOrder}(H', Ts', TT', i, j)$, and $\mathsf{PartialThreadOrder}(H, Ts, TT)$, there is no sequence of threads granting capabilities to each others' locks starting with a lock held by thread j and ending with the first lock acquired by thread i. So there is no tentative edge ending at thread i's first lock, and this case cannot introduce a multi-thread cycle.
    - The path ends with a tentative edge to another lock held by thread i. For this to be the case, thread i must have orphaned a lock, which can't be the case because by the lemma results above thread i has no orphaned locks because $L \neq []$.

  Thus, $\mathsf{SafeCapUse}(J', G')$.

  In all cases for E-VAR, the $\mathsf{PartialThreadOrder}$ assertion is preserved because the capability-granting relation and the locations of unique references are unaffected.
- Case E-FIELD: Similar to the previous case.
- Case E-WVAR: $H, i, V_i, Ls_i, x := v \to H, V_i[x \mapsto v], Ls_i, v$. By Lemma 4, there exists Σ', TT' such that

  – $P \vdash H'; Ts' : \Sigma'; TT'$

  $Ls_i = Ls_i'$, so $G' = \mathsf{StepGraph}(G, Ts, Ts') = G$. It is straightforward to apply G-MATCH-REDUCTION to derive:

  – $G' \equiv Ts'$

  This evaluation step produces a value, and the same argument for producing $\mathsf{SafeCapUse}(J', G')$ applies as in the E-VAR case. The remaining proof obligation is $\mathsf{PartialThreadOrder}(H', Ts')$. We prove it by inversion on the typing derivation for the initial state, and by case analysis on $\Gamma_i(x)$ as used in T-WVAR:

  – $\Gamma_i(x) = \text{final } \tau$: By induction on the program typing and the thread typing derivation, this was not the case.
  – $\Gamma_i(x) = \text{guardless } c$: This case stores a unique reference into a local variable, which affects thread ordering in two ways. First, it may overwrite a non-null unique reference, which may either reduce or preserves the thread ordering. Second, it moves a unique reference from an expression literal in e_i to a local variable in V_i'. This preserves the ordering of thread i with respect to any locks capability-reachable from v because it changes the location from satisfying the third clause of $\mathsf{ThreadOrder}$ to satisfying the second, if it didn't already satisfy the first clause. Thus because the relative ordering of all threads is preserved or removed, $\mathsf{PartialThreadOrder}(H', Ts', TT')$.
  – $\Gamma_i(x) = \text{partial } c$ or $\Gamma_i(x) = \text{borrowed}\langle y \rangle\ c$: These cases do not affect the $\mathsf{PartialThreadOrder}$ assumptions of the previous state, so they are trivially preserved.
- Case E-WFIELD: Similar to the previous case, except that storing to a unique field may remove capability-reachability for the overwritten value instead of simply removing its satisfaction of clauses in $\mathsf{ThreadOrder}$, and similarly with the storage of a unique value changing that value from satisfying the third clause to the first clause of $\mathsf{ThreadOrder}$.
- Case E-LOCK: By Lemma 4, there exists Σ', TT' such that

  – $P \vdash H'; Ts' : \Sigma'; TT'$

  By Lemma 5:

  – $\Sigma; \phi_i; \Upsilon_{i,s}; \Gamma_i; L \vdash \text{lock } l\ e_b : \tau'; \Upsilon_i''$

- $L = [] \vee (\phi_i(l) = x \wedge V_i(x) = l_x \wedge x \in L \wedge (\forall z \in L : L = L'@[z] \vee \exists p'.\mathsf{RaceFreePath}(\Gamma_i, L, p') \wedge \mathsf{FinalAlias}(\Gamma_i, p', z) \wedge \Gamma_i; L \vdash p' : \text{complete}\langle a \rangle\ c))$
- $\Gamma_i; V_i \vdash Ls_i : L$

By induction on L:

- Case $L = []$: By the last lemma result $Ls_i = []$, so $G' = \mathsf{StepGraph}(G, Ts, Ts') = G[i \mapsto (\{l\}, [])]$. It is straightforward to apply G-MATCH-REDUCTION to derive:
  - $G' \equiv Ts'$

  This adds no new edges to the joint graph J', so $\mathsf{SafeCapUse}(J', G')$. This also changes thread ordering, placing thread i between any thread that might control the unique reference to l and any threads whose first-acquired lock are capability-reachable from l. The ordering remains acyclic, so $\mathsf{PartialThreadOrder}(H', Ts', TT')$.
- Case $L = l_o :: L_o$: By the lemma results above, this case adds to the joint graph J' an edge that would have been an intended edge in J. Because the vertices of G match the held locks in the initial state by $G \equiv Ts$, and because by the lemma results the new edge was reflected in the capability-granting relation, there are not outgoing edges from l in J' and by inversion on the program typing for the initial state, no other incoming edges. So $\mathsf{SafeCapUse}(J', G')$. Anything effectively capability-reachable from thread i before is still effectively capability-reachable, in some cases simply beginning from a different lock. So $\mathsf{PartialThreadOrder}(H', Ts', TT')$. It is straightforward to apply G-MATCH-REDUCTION to derive:
  - $G' \equiv Ts'$

  Note that this reduction does not produce a value, and therefore does not reduce the thread to a state that imposes a new intended edge in J'.

- Case E-UNLOCK: This case is straightforward; it may add a new tentative edge to the joint graph, but otherwise removes an edge from G and J, preserving multi-thread acyclicity of the joint graph, and potentially removing itself from the thread ordering relation if it releases its only lock.
- Case E-RECLOCK: This case is straightforward; the additional edge in G' and J' is between two locks held by the same thread, and may therefore only complete a multi-thread cycle in J' if there was already such a cycle in J, which contradicts the inductive hypothesis because we know that J lacked any multiple thread cycles when the edge added to G was present in J as a tentative edge. It also does not affect thread ordering because the reduced thread holds the same set of locks, and it cannot reduce the thread to a state that imposes a new intended edge.
- Case E-RECUNLOCK: Similar to the E-UNLOCK case above.
- Case E-DVAR: Similar to the E-WVAR case, but with the unique reference moving the opposite direction, and no overwriting can occur.
- Case E-DFIELD: By Lemma 4, there exists Σ', TT' such that
  - $P \vdash H'; Ts' : \Sigma'; TT'$

  $Ls_i = Ls_i'$, so $G' = \mathsf{StepGraph}(G, Ts, Ts') = G$. It is straightforward to apply G-MATCH-REDUCTION to derive:
  - $G' \equiv Ts'$

  This reduction does not change the effective thread ordering, because those threads whose first locks were reachable from the mutated (locked) object are now reachable from a unique reference present as a literal in e_i'.

  This evaluation step can produce a value, and therefore could introduce an intended edge in $J' = \mathsf{Join}(G', TT', Ts')$; the argument for proving $\mathsf{SafeCapUse}(J', G')$ is the same as demonstrated in the E-VAR case.
- Case E-SPAWN: By Lemma 4, there exists Σ', TT' such that
  - $P \vdash H'; Ts' : \Sigma'; TT'$

  Because no lock sets changed from Ts to Ts' but a new thread was created, $G' = \mathsf{StepGraph}(G, Ts, Ts') = G$. By inversion on the new state's program typing, the newly created thread j holds no locks ($Ls_j' = []$). By inversion on the program typing for the new state and induction on the reduced thread's (i) typing derivation, the x passed to the new thread had a particular type $\Gamma_i(x)$. By case analysis on the type:
  - $\Gamma_i(x) \neq \text{guardless}\ c$: Thread ordering is unaffected.
  - $\Gamma_i(x) = \text{guardless}\ c$: Thread ordering strictly decreases, because thread i will no longer be ordered before any threads holding locks capability-reachable from $V_i(x)$.

  Either way, $\mathsf{PartialThreadOrder}(H', Ts', TT')$. Because the reduction in thread i cannot produce a non-null value as the target of a lock expression, $J' = J$, so $\mathsf{SafeCapUse}(J', G')$.

□

Lemma 7 (Deadlock Freedom)**.** *If* $\vdash P$ *and* $(\emptyset, \{(tid_0, \emptyset, [], \text{Expression}(P))\}) \rightarrow^* (H, Ts)$*, then* (H, Ts) *is not deadlocked.*

Proof. A state is deadlocked if there is some cycle of threads such that each is blocked trying to acquire a lock held by the next thread in the cycle. This manifests in the joint capability use graph of a state as a path between two locks held by the same thread that uses edges from multiple threads. That is precisely one of the conditions that Lemma 6 preserves the absence of, and since the initial program state clearly satisfies the criteria to apply both preservation lemmas (4 and 6), any state reachable must not be deadlocked. □