Higher-Order Leak and Deadlock Free Locks

JULES JACOBS, Radboud University, The Netherlands
STEPHANIE BALZER, Carnegie Mellon University, USA

Reasoning about concurrent programs is challenging, especially if data is shared among threads. Program correctness can be violated by the presence of data races—whose prevention has been a topic of concern both in research and in practice. The Rust programming language is a prime example, putting the slogan fearless concurrency in practice by not only employing an ownership-based type system for memory management, but also using its type system to enforce mutual exclusion on shared data. Locking, unfortunately, not only comes at the price of *deadlocks* but shared access to data may also cause memory *leaks*.

This paper develops a theory of deadlock and leak freedom for *higher-order locks* in a shared memory concurrent setting. Higher-order locks allow sharing not only of basic values but also of other locks and channels, and are themselves first-class citizens. The theory is based on the notion of a *sharing topology*, administrating who is permitted to access shared data at what point in the program. The paper first develops higher-order locks for *acyclic* sharing topologies, instantiated in a λ-calculus with higher-order locks and message-passing concurrency. The paper then extends the calculus to support *circular* dependencies with *dynamic* lock orders, which we illustrate with a dynamic version of Dijkstra's dining philosophers problem. Well-typed programs in the resulting calculi are shown to be free of deadlocks and memory leaks, with proofs mechanized in the Coq proof assistant.

CCS Concepts: • **Software and its engineering → Concurrent programming languages**.

Additional Key Words and Phrases: Concurrency, Higher-Order Lock, Deadlock, Memory Leak

ACM Reference Format:
Jules Jacobs and Stephanie Balzer. 2023. Higher-Order Leak and Deadlock Free Locks. *Proc. ACM Program. Lang.* 7, POPL, Article 36 (January 2023), 31 pages. https://doi.org/10.1145/3571229

1 INTRODUCTION

Today's applications are inherently concurrent, necessitating programming languages and constructs that support spawning of threads and *sharing* of resources. Sharing of resources among threads, a sine qua non for many applications, is the source of many concurrency-related software bugs. The issue is the possibility of a race condition, if simultaneous write and read accesses are performed to shared data. To rule out data races, locks can be employed, forcing simultaneous accesses to happen in mutual exclusion from each other. Locking unfortunately not only comes at the cost of *deadlocks*, but shared access to data may also cause *memory leaks*.

This paper develops a *λ-calculus* with *higher-order locks* and message-passing concurrency, where well-typed programs are free of memory leaks and deadlocks. Whereas there exist type systems for memory safety—most notably Rust [Jung et al. 2018a; Matsakis and Klock 2014], incorporating ideas of ownership types [Clarke et al. 1998; Müller 2002] and region management [Grossman et al. 2002; Tofte and Talpin 1997]—memory safety only ensures that no dangling pointers are dereferenced, but does not rule out memory leaks. Similarly, type systems for deadlock and leak

Authors' addresses: Jules Jacobs, mail@julesjacobs.com, Radboud University, Nijmegen, The Netherlands; Stephanie Balzer, balzers@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, USA.

2475-1421/2023/1-ART36
https://doi.org/10.1145/3571229

freedom have been developed, pioneered by Igarashi and Kobayashi [1997]; Kobayashi [1997]; Kobayashi et al. [1999] in the context of the π-calculus and by Caires and Pfenning [2010]; Wadler [2012] in the context of linear logic session types. Our work builds on the latter and extends it with the capability to share resources as present in a shared memory setting.

It may come as a surprise that locking not only can cause deadlocks but also memory leaks. We give an example of a memory leak caused by a mutex in Rust below:

```
struct X { x: Option<Arc<Mutex<X>>> } // declare type that will be stored in the mutex

let m1 = Arc::new(Mutex::new(X { x: None })); // create mutex with empty payload
let m2 = m1.clone(); // create a second reference to the mutex, incrementing refcount
let mut g = m1.lock(); // acquire the mutex, giving access to the contents
*g = X { x: Some(m2) }; // mutate the contents to store m2 in the mutex
drop(g); // release the lock
drop(m1); // drop the reference to the mutex, decrementing the refcount
```

On the first line, we declare a recursive struct, that optionally contains a reference to a mutex that is reference counted. On the second line, we then create such a mutex, initially with empty payload. We then clone the reference to the mutex, raising the reference count to 2. Finally, we lock the mutex through the first reference and store the second reference in it, transferring ownership of `m2` to the mutex. On the last line, we release the mutex and drop the reference to `m1`. This decrements the reference count to 1, but there still exists a self-reference from inside the mutex, leading to a memory leak.

It is tempting to conclude from the above example that recursive types are necessary to create memory leaks. This is not the case, however. Instead of storing the mutex inside the mutex directly, one can store a closure of type `unit → unit` that captures the mutex in its lexical environment.

Memory leaks can moreover be caused by channels, as illustrated by the below Rust code:

```
struct Y { y: Receiver<Y> } // declare type that will be sent over the channel

let (s,r) = mpsc::channel(); // create a channel with sender s and receiver r
s.send(Y { y: r }); // put the receiver in the buffer of the channel
drop(s); // drop the reference to the sender; but memory is leaked
```

On the first line, we declare a recursive struct with a reference to a receiver endpoint of a channel. On the second line, we then allocate a channel, which gives us a sender `s` and receiver `r`. We then send the receiver along the sender, transferring it into the channel's buffer. When we drop the sender, the reference to the receiver still exists from within the buffer, creating a memory leak.

Unsurprisingly, we can use the same concurrency constructs to also cause deadlocks. For example, a thread my allocate a new channel, keep both the sender and the receiver reference, and attempt to receive from the receiver before sending along the sender:

```
let (s,r) = mpsc::channel(); // create a new channel
r.recv(); // this call blocks on receiving a message, deadlock!
s.send(3); // the message is sent, but too late
```

Similarly, mutexes can give rise to deadlocks. Consider the following `swap` function:

```
fn swap(m1: &Mutex<i32>, m2: &Mutex<i32>) {
  let mut g1 = m1.lock(); // acquire first mutex
  let mut g2 = m2.lock(); // acquire second mutex
  let tmp = *g1; // obtain the contents stored in m1
```

```
  *g1 = *g2; // replace the contents of m1 with the contents of m2
  *g2 = tmp; // replace the contents of m2 with the original contents of m1
  drop(g1); drop(g2) // release the locks
}
```

This function takes two references to mutexes, locks both, and swaps their contents. Now let's consider the below code that calls this function:

```
let m1 = Arc::new(Mutex::new(1)); // create a new mutex
let m2 = m1.clone(); // create a second reference to the mutex
swap(&m1,&m2); // deadlock!
```

The code allocates a mutex, yielding the reference `m1`, and then creates an alias `m2` to the same mutex. Then it calls function `swap` with `m1` and `m2` as arguments. The function will deadlock upon the second acquire, which will block until the first one is released. This last example also demonstrates that reasoning about deadlocks—and for that matter memory leaks—is not inherently local. Both the function `swap` and the above code are benign on their own but problematic when composed.

The above examples use the API constructs `Arc<Mutex<T>>` and `Rc<RefCell<T>>` to cause memory leaks and deadlocks, suggesting that substructural typing is insufficient to rule out memory leaks and deadlocks, but that memory leak and deadlock freedom must be accounted for at the level of API design. Based on this observation, we develop a λ-calculus for shared memory concurrency with a lock data type, guaranteeing absence of *memory leaks* and *deadlocks* by type checking. Memory leaks are especially bothersome for resource-intensive applications, and deadlocks can prevent an entire application from being productive. We phrase our lock API and type system in a λ-calculus setting to keep it independent of an actual target language, yet readily adoptable by any language with similar features. Locks in our calculus are *higher-order*, allowing them to store not only basic values but also other locks. This feature enables us to encode *session typed channels* [Honda 1993; Honda et al. 1998]. These channels are also higher-order, and can thus be stored in locks and sent over each other as well.

While higher-order locks and channels increase the expressivity of our calculus and scale it to realistic application scenarios, they also challenge our goal to ensure deadlock and leak freedom by type checking. Our approach is to account for an application's *sharing topology*, which tracks, for every lock, *(i)* who has references to the lock, *(ii)* who is responsible for releasing the lock, and *(iii)* who is responsible for deallocating the lock. The fundamental invariant that we place on the sharing topology demands that there never exist any circular dependencies among these responsibilities at any point in the execution of a program.

We first develop the calculus λ_{lock}, which enforces this invariant preemptively, by demanding that the sharing topology be *acyclic*. As a result, λ_{lock} enjoys memory leak and deadlock freedom, with corresponding theorems and proofs developed in Section 4 and Section 5, respectively. We then introduce the calculus $\lambda_{\text{lock}^{++}}$, an extension of λ_{lock} that supports circular resource dependencies, as famously portrayed by Dijkstra's dining philosophers problem, while preserving memory leak and deadlock freedom. $\lambda_{\text{lock}^{++}}$ permits *cyclic* sharing dependencies within *lock groups* using a *lock order*, but satisfies the sharing topology's fundamental invariant between different lock groups. These orders are purely local to a lock group and can change dynamically by the addition or removal of locks to and from a group. Local orders are compositional in that they remove the need for local orders to comply with each other or a global lock order when acquiring locks from distinct groups. The proofs of memory leak and deadlock freedom for λ_{lock} and $\lambda_{\text{lock}^{++}}$ are mechanized in the Coq proof assistant and detailed in Section 7.

In summary, this paper contributes.

- A notion of acyclic sharing topology to rule out circular dependencies without any restriction on the order in which operations must be performed,
- the language λ_{lock} with higher-order locks for shared memory concurrency and type system based on the sharing topology to ensure memory leak freedom and deadlock freedom,
- an encoding of session-typed message-passing channels in terms of locks,
- the language $\lambda_{\text{lock++}}$, an extension supporting cyclic unbounded process networks,
- proofs of deadlock and memory leak freedom for well-typed λ_{lock} and $\lambda_{\text{lock++}}$ programs, mechanized in Coq.

2 KEY IDEAS AND EXAMPLES

This section develops the notion of a *sharing topology* underlying our calculus and illustrates its type system based on examples. We start by deriving the fundamental invariant to be preserved by the sharing topology in several steps, distilling several key principles. We first focus on *acyclic* sharing topologies, resulting in the calculus λ_{lock}, which we then extend to account for *cyclic* sharing dependencies, resulting in the calculus $\lambda_{\text{lock++}}$.

2.1 Invariant for leak and deadlock freedom

The examples of memory leaks and deadlocks discussed in Section 1 all share a common pattern: a thread has several references to the same lock, introducing self-referential responsibilities for releasing and deallocating locks. **Our goal is to devise a system that allows threads to reason *locally* about shared resources and, in particular, to give threads complete freedom to acquire and release any lock they reference.** Our approach thus opts for restricting the propagation of lock references by an individual thread rather than their use. To forbid the self-referential scenarios discussed in Section 1, the fundamental invariant of the sharing topology must satisfy the following principle:

Principle 1: Each thread only holds one reference to any given lock.

To satisfy this principle our calculus treats locks *linearly*, ensuring that references to locks cannot be duplicated *within* a thread.

The above principle is obviously not yet sufficient for ruling out deadlocks, as deadlocks can also result when two threads compete for resources. For example, consider two threads T_1 and T_2 with references to locks l_1 and l_2. A deadlock can arise if thread T_1 tries to acquire lock l_1 and then l_2, and thread T_2 tries to acquire lock l_2 and then lock l_1. Therefore, the fundamental invariant must also satisfy the following principle (which Section 6 relaxes by permitting sharing of a group of locks rather than an individual lock):

Principle 2: Any two threads may share at most one lock.

This principle is still not yet sufficient for ruling out deadlocks. Consider an example with 3 threads T_1,T_2,T_3, and 3 locks l_1,l_2,l_3, where:

- thread T_1 acquires l_1 and then l_2
- thread T_2 acquires l_2 and then l_3
- thread T_3 acquires l_3 and then l_1

If a schedule allows each thread to acquire their first lock, the threads will subsequently deadlock when trying to acquire their second lock. Note, however, that the preceding principle is satisfied: thread T_1 and T_2 only share lock l_2, thread T_2 and T_3 only share lock l_3, and tread T_3 and T_1 only share lock l_1. Thus, if we want to uphold thread local reasoning, while guaranteeing that thread composition preserves deadlock freedom, we must impose a stronger invariant, constraining the sharing topology:

Principle 3: If we consider the graph where threads are connected to the locks they hold a reference to, this graph must not have a cycle.

Our calculus enforces this principle by the following lock and thread operations, which bear a resemblance to channels in linear logic session types based on cut elimination [Caires and Pfenning 2010; Wadler 2012]:

- **new** to create a new lock. Threads are free to use this operation. Because the lock is created, the creating thread is the only one to have a reference to it.
- **acquire** to acquire a lock. This operation can be called at any time, but the type system must ensure that the same lock cannot be acquired multiple times via the same reference.
- **release** to release the lock. This operation can be called at any time, and the type system *must* ensure that it is called eventually for any acquired lock.
- **fork**, which forks off a new thread, and allows the programmer to create a new reference to *one* lock from the parent thread and share it with the child thread.

Although the **fork** construct allows duplicating a lock reference, the newly created reference must be passed to the forked off thread, creating a new edge between the new thread and the lock. If we restrict sharing of a lock between a parent and child thread to exactly *one* lock, the graph arising from the reference structure between threads and locks remains acyclic. For example, consider the threads T_1,T_2,T_3 and locks l_1 and l_2 such that:

- threads T_1 and T_2 share lock l_1
- threads T_1 and T_3 share lock l_2

If T_1 spawns T_4 and provides a reference to l_1, the resulting reference structure remains acyclic: T_1 is connected to l_1 and l_2, with the former being connected to T_2 and T_4 and the latter being connected to T_3. However, if we allowed T_1 to share both l_1 and l_2 with T_4, the graph becomes cyclic: T_1 is connected to T_4 both via l_1 and l_2.

The type system that we sketch in the next section and detail in Section 3 enforces the above rules and thus upholds the principles derived so far to rule out deadlocks. A reader may wonder whether these principles are strong enough for asserting deadlock freedom in the presence of higher-order locks, allowing us to store locks in locks. For example, we can easily transfer a lock l_1 from thread T_1 to thread T_2 by storing it in a lock l_2 shared between T_1 and T_2, allowing T_2 to retrieve l_1 by acquiring l_2. This scenario is indeed possible and turns out not to be a problem. While not immediately obvious, this transfer actually preserves acyclicity of the sharing topology. To account for the possibility of references between locks, we refine our invariant as follows:

Principle 4: If we consider the graph where threads are connected to the locks they hold a reference to, and locks are connected to locks they hold a reference to, this graph must not have a cycle.

Principle 4 amounts to an invariant that is sufficient to ensure deadlock freedom. Section 5 details that a well-typed λ_{lock} program preserves this invariant along transitions. The next question to explore is whether this invariant is also sufficient to ensure memory leak freedom.

It seems that the above invariant is sufficient for ruling out the examples of memory leaks portrayed in Section 1, because they are all instances of self-referential structures, prevented by the above invariant. However, to answer this question entirely, we have to remind ourselves of our definition of a sharing topology given in Section 1:

Definition 2.1. A *sharing topology* tracks, for every lock, *(i)* who has references to the lock, *(ii)* who is responsible for releasing the lock, and *(iii)* who is responsible for deallocating the lock.

So far, we have only accommodated the first two ingredients, but yet have to establish responsibility of lock deallocation.

To get started, let us first explore the question of *"how to ever safely get rid of a lock"*. Obviously, we should not attempt to drop a reference to a lock that we have acquired, because then this lock would never be released, blocking any other threads that are trying to acquire that lock. So, is it then safe to drop a reference to a lock that we have *not* currently acquired? As a matter of fact even this is not safe, if we allow storing linear data in locks. For example, we could then easily discard a linear value v as follows, which would defeat the purpose of linear typing:

(1) Create a new lock and acquire it.
(2) Put the linear value v in the lock.
(3) Release the lock.
(4) Drop the reference to the lock.

We thus face the following conundrum: if we allow dropping references to an acquired lock, then we cannot leak data, but we get deadlocks, and if we allow dropping references to a non-acquired lock, then we can leak data (which then allows us to create deadlocks anyway).

It seems that we have to circle back to Definition 2.1 and find a way to designate one reference among all the references to a lock as the one that carries the responsibility for deallocation. For this purpose we differentiate lock references into an *owning reference*, which carries the responsibility to deallocate the lock, and *client references*, which can be dropped. Naturally, there must exist exactly one owning reference. An owning reference can only be dropped after the lock has been deallocated. To deallocate the lock, the owner must first *wait* for all the clients to drop their references and then retrieve the contents of the lock.

This brings us to our final invariant:

Principle 5: If we consider the graph where threads are connected to the locks they hold a reference to, and locks are connected to locks they hold a reference to, this graph must not have a cycle. Furthermore, each lock must have precisely one owning reference, and zero or more client references.

2.2 The $\mathbf{Lock}\langle\tau\,{}^{a}_{b}\rangle$ data type and its operations

Let us now investigate what a lock API and type system based on these principles look like. A detailed discussion of the resulting language λ_{lock} is given in Section 3.

We introduce the following type of lock references:

$$\mathbf{Lock}\langle\tau\,{}^{a}_{b}\rangle$$

where

- $\tau \in \mathbf{Type}$ is the type of values stored in the lock.
- $a \in \{0, 1\}$ indicates whether this reference is the owner ($a = 1$) or a client ($a = 0$).
- $b \in \{0, 1\}$ indicates whether this reference has acquired the lock ($b = 1$) or not ($b = 0$).

λ_{lock} supports the following operations to acquire and release locks:

$$\mathbf{acquire} : \mathbf{Lock}\langle\tau\,{}^{a}_{0}\rangle \to \mathbf{Lock}\langle\tau\,{}^{a}_{1}\rangle \times \tau$$
$$\mathbf{release} : \mathbf{Lock}\langle\tau\,{}^{a}_{1}\rangle \times \tau \to \mathbf{Lock}\langle\tau\,{}^{a}_{0}\rangle$$

These operations are *linear* and hence consume their argument. Both operations return the lock argument reference at a different type, reflecting whether the lock is currently acquired or not. The acquire operation gives the user full access to the τ value protected by the lock, and the release operation requires the user to put back a τ value. Acquire and release operations work for $a \in \{0, 1\}$,

so both clients and the owner are allowed to acquire and release the lock. We find it helpful to think of a lock as a shared "locker" or container to exchange valuables. Using this metaphor, we can perceive an acquire as opening the closed locker to retrieve the valuable and a release as closing an open locker to store the valuable. If a reference ℓ is of type $\mathbf{Lock}\langle \tau\, {}^{a}_{1}\rangle$, indicating that the locker has been opened, the holder of the reference is responsible for eventually putting back the valuable using a **release**. If a reference ℓ is of type $\mathbf{Lock}\langle \tau\, {}^{a}_{0}\rangle$, indicating that the locker has not been opened via the reference ℓ, the holder of the reference is allowed to try to acquire the locker.

Let us now look at how locks are created and destroyed. We have three operations, one for creating a lock, one for deallocating a lock via its owning reference, and one for dropping a client reference to a lock:

$$\mathbf{new} : \mathbf{1} \to \mathbf{Lock}\langle \tau\, {}^{1}_{1}\rangle$$
$$\mathbf{wait} : \mathbf{Lock}\langle \tau\, {}^{1}_{0}\rangle \to \tau$$
$$\mathbf{drop} : \mathbf{Lock}\langle \tau\, {}^{0}_{0}\rangle \to \mathbf{1}$$

The operation **new** creates an owning reference. The operation **wait** on the owning reference waits for all clients to finish and then returns ownership of the value τ stored in the lock (and frees the memory associated with the lock). The operation **drop** on a client reference yields unit, effectively not returning anything. The drop operation could potentially be automatically inserted by a compiler, as it is done by the Rust compiler, for example, but we prefer to be explicit. Note that both **wait** and **drop** require the lock to be in a non-acquired (*a.k.a.,* closed) state, which means that a thread holding an open lock reference must fulfill its obligation to put a value back into the lock using **release** before it is allowed to use **drop** or **wait** on that lock reference. This ensures that **drop** and **wait** cannot cause another thread's **acquire** to deadlock. The details of deadlock freedom can be found in Sections 4 and 5.

Client references are created upon fork:

$$\mathbf{fork} : \ \mathbf{Lock}\langle \tau\, {}^{a_1+a_2}_{b_1+b_2}\rangle \times (\mathbf{Lock}\langle \tau\, {}^{a_2}_{b_2}\rangle \multimap \mathbf{1}) \ \ \to \ \ \mathbf{Lock}\langle \tau\, {}^{a_1}_{b_1}\rangle$$

It may be helpful to consider an example, where ℓ has type $\mathbf{Lock}\langle \tau\, {}^{a_1+a_2}_{b_1+b_2}\rangle$:

$$\mathbf{let}\ \ell_1 : \mathbf{Lock}\langle \tau\, {}^{a_1}_{b_1}\rangle \ = \ \mathbf{fork}(\ell,\ \lambda \ell_2 : \mathbf{Lock}\langle \tau\, {}^{a_2}_{b_2}\rangle.\ (\cdots))$$

The fork operation consumes the original lock reference ℓ, and splits it into two references, ℓ_1 and ℓ_2. The reference ℓ_1 is returned to the main thread, and the reference ℓ_2 is passed to the child thread. The child thread runs the code indicated by $(\cdots)$, which has access to ℓ_2. Therefore, in terms of types,

$$\mathbf{Lock}\langle \tau\, {}^{a_1+a_2}_{b_1+b_2}\rangle \quad \text{is split into} \quad \begin{cases} \mathbf{Lock}\langle \tau\, {}^{a_1}_{b_1}\rangle \\ \mathbf{Lock}\langle \tau\, {}^{a_2}_{b_2}\rangle \end{cases}$$

such that $a_1 + a_2 \leq 1$ and $b_1 + b_2 \leq 1$. This side condition ensures that, if the original lock reference ℓ is an owner reference and thus of type $\mathbf{Lock}\langle \tau\, {}^{1}_{b}\rangle$, it can only be split into an owner reference and client reference. Conversely, if the original lock reference ℓ is a client reference and thus of type $\mathbf{Lock}\langle \tau\, {}^{0}_{b}\rangle$, it can only be split into two client references. Similarly, if the original reference ℓ is acquired and thus of type $\mathbf{Lock}\langle \tau\, {}^{a}_{1}\rangle$, only one of the new references is acquired, and if the original reference ℓ is not acquired and thus of type $\mathbf{Lock}\langle \tau\, {}^{a}_{0}\rangle$, both of the new references are not acquired either.

The standard rules of binding and scope apply to the lambda used in a **fork** as well. For example, we can transfer linear resources from the main thread to a child thread, *e.g.,* the resource bound to

the linear variable r in the following example:

$$\begin{array}{l}\textbf{let } \ell \;=\; \textbf{new}() \textbf{ in} \\ \textbf{let } r \;=\; \textbf{new}() \textbf{ in} \\ \textbf{let } \ell_1 \;=\; \textbf{fork}(l, \lambda \ell_2.\ (\cdots r \cdots)) \textbf{ in } (\cdots)\end{array}$$

Here, the resources r can no longer be used in the main thread because of linearity.

These are all the constructs of λ_{lock} that concern locks. Section 3 details how we integrate λ_{lock} with session-typed channels, facilitating the exchange of locks between threads not only by storing them into other locks, but also by sending them along channels, possibly as part of a compound data structure. Channels, of course, are first-class as well, allowing them to be sent over each other and stored in locks. Given the range of possibilities of how the sharing topology of a program can change dynamically, a reader may be surprised that λ_{lock} asserts memory leak and deadlock freedom by type checking. After all, as usual, the devil is in the details! The formal statement of memory leak and deadlock freedom is given in Section 4, and their proof is sketched in Section 5. For the full details, the reader is referred to the mechanization Section 7.

2.3 Examples

We now look at a few examples that illustrate the use of lock operations.

2.3.1 Locks as mutable references. A lock without any client references can be viewed as a linear mutable reference cell. We can create such a reference cell with $\ell = \textbf{release}(\textbf{new}(), v)$, read its value with $\textbf{acquire}(\ell)$, and write into it a new value with $\textbf{release}(\ell, v)$. We can also deallocate the reference with $\textbf{wait}(\ell)$, which gives us back the value.

$$\begin{array}{l}\textbf{let } \ell \;=\; \textbf{release}(\textbf{new}(), 1) \textbf{ in} \\ \textbf{let } \ell, n \;=\; \textbf{acquire}(\ell) \textbf{ in} \\ \textbf{let } \ell \;=\; \textbf{release}(\ell, n+1) \textbf{ in} \\ \textbf{let } m \;=\; \textbf{wait}(\ell)\end{array}$$

If the values we store in the cell are of unrestricted type (duplicable and droppable), we can implement references with **ref**, **get**, and **set** as follows:

$$\begin{array}{rl}\textbf{ref} : \tau \to \textbf{Lock}\langle \tau\, {}^{1}_{0} \rangle & \\ \textbf{ref}(v) \triangleq \textbf{release}(\textbf{new}(), v) & \end{array}$$

$$\begin{array}{rl}\textbf{get} : \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle \to \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle \times \tau & \text{where } \tau \text{ unr} \\ \textbf{get}(\ell) \triangleq \textbf{let } \ell, v \;=\; \textbf{acquire}(\ell) \textbf{ in } (\textbf{release}(\ell, v), v) & \end{array}$$

$$\begin{array}{rl}\textbf{set} : \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle \times \tau \to \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle & \text{where } \tau \text{ unr} \\ \textbf{set}(\ell, v) \triangleq \textbf{let } \ell, v' \;=\; \textbf{acquire}(\ell) \textbf{ in } \textbf{release}(\ell, v) & \end{array}$$

Similarly, we can atomically exchange the value as follows or apply a function to the value as follows:

$$\begin{array}{rl}\textbf{xchng} : \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle \times \tau \to \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle \times \tau & \\ \textbf{xchng}(\ell, v) \triangleq \textbf{let } \ell, v' \;=\; \textbf{acquire}(\ell) \textbf{ in } (\textbf{release}(\ell, v), v') & \end{array}$$

$$\begin{array}{rl}\textbf{modify} : \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle \times (\tau \multimap \tau) \to \textbf{Lock}\langle \tau\, {}^{a}_{0} \rangle & \\ \textbf{modify}(\ell, f) \triangleq \textbf{let } \ell, v \;=\; \textbf{acquire}(\ell) \textbf{ in } \textbf{release}(\ell, f\ v) & \end{array}$$

These operations work even for linear values, since neither v nor v' are duplicated or dropped.

2.3.2 Fork-join / futures / promises. In the generalised fork-join model with futures / promises, the parent thread can spawn a child thread to do some work, and later synchronize with the child to obtain the result. Our lock operations directly support this:

- The parent thread creates a new lock using $\ell = \mathbf{new}()$.
- The parent forks off the child thread, sharing an opened client reference to ℓ with the child.
- The parent thread continues doing other work, and eventually calls $\mathbf{wait}(\ell)$ on the lock.
- When the child thread is done with the work, it calls $\mathbf{release}(\ell, v)$ with the result v.

This is illustrated in the following program:

$$\begin{array}{l}\mathbf{let}\ \ell\ =\ \mathbf{fork}(\mathbf{new}(), \lambda\ell.\ \cdots \mathbf{drop}(\mathbf{release}(\ell, v))\cdots)\ \mathbf{in}\\ \cdots\\ \mathbf{let}\ v\ =\ \mathbf{wait}(\ell)\end{array}$$

Note how the type system ensures deadlock and leak freedom:

- Initially, $\mathbf{new} : \mathbf{Lock}\langle\tau\ {}^{1}_{1}\rangle$, *i.e.,* the lock is an open owner reference.
- When we fork, we split the lock up into $\mathbf{Lock}\langle\tau\ {}^{1}_{0}\rangle$ and $\mathbf{Lock}\langle\tau\ {}^{0}_{1}\rangle$.
- The closed and owning reference of type $\mathbf{Lock}\langle\tau\ {}^{1}_{0}\rangle$ goes to the parent, who eventually waits for the result.
- The open and client reference of type $\mathbf{Lock}\langle\tau\ {}^{0}_{1}\rangle$ goes to the child, who must put a value in it in order to drop it.

Of course, the client is free to pass around its reference to the lock, which acts as a future/promise, so that somebody else can fulfill the obligation to release the lock by putting a value in it.

2.3.3 Concurrently shared data. The parent thread can spawn multiple child threads and create a new lock for each, as in the fork-join pattern. However, the parent can also create one lock, put an initial data structure v in it, and share lock references with several children, who may each acquire and mutate the lock's contents repeatedly:

$$\begin{array}{l}\mathbf{let}\ \ell\ =\ \mathbf{release}(\mathbf{new}(), v)\ \mathbf{in}\\ \mathbf{let}\ \ell\ =\ \mathbf{fork}(\ell, \lambda\ell.\ \cdots)\ \mathbf{in}\\ \mathbf{let}\ \ell\ =\ \mathbf{fork}(\ell, \lambda\ell.\ \cdots)\ \mathbf{in}\\ \mathbf{let}\ \ell\ =\ \mathbf{fork}(\ell, \lambda\ell.\ \cdots)\ \mathbf{in}\\ \cdots\\ \mathbf{let}\ v'\ =\ \mathbf{wait}(\ell)\end{array}$$

Children are of course free to fork off children of their own, all sharing access to the same lock ℓ.

2.3.4 Bank example. Consider three bank accounts whose balances are stored in locks ℓ_1, ℓ_2, ℓ_3. The main thread acts as the bank, spawns three clients, and gives them access to their bank account so that they can deposit and withdraw money from it:

$$\begin{array}{l}\mathbf{let}\ \ell_1\ =\ \mathbf{fork}(\mathbf{release}(\mathbf{new}(), 0), \lambda\ell_1.\ \cdots \mathit{client}\ 1\cdots)\ \mathbf{in}\\ \mathbf{let}\ \ell_2\ =\ \mathbf{fork}(\mathbf{release}(\mathbf{new}(), 0), \lambda\ell_2.\ \cdots \mathit{client}\ 2\cdots)\ \mathbf{in}\\ \mathbf{let}\ \ell_3\ =\ \mathbf{fork}(\mathbf{release}(\mathbf{new}(), 0), \lambda\ell_3.\ \cdots \mathit{client}\ 3\cdots)\ \mathbf{in}\\ \cdots\\ \mathbf{let}\ \ell_1, \ell_2\ =\ \mathsf{transaction}(\ell_1, \ell_2, 50)\ \mathbf{in}\ \cdots\end{array}$$

The bank does a transaction between ℓ_1 and ℓ_2:

$$\begin{aligned}
&\mathsf{transaction} : \mathbf{Lock}\langle \mathbf{int}\,{}^{a}_{0}\rangle \times \mathbf{Lock}\langle \mathbf{int}\,{}^{a}_{0}\rangle \multimap \mathbf{Lock}\langle \mathbf{int}\,{}^{a}_{0}\rangle \times \mathbf{Lock}\langle \mathbf{int}\,{}^{a}_{0}\rangle \\
&\mathsf{transaction}(\ell_1, \ell_2, \mathsf{amount}) \triangleq \\
&\quad \textbf{let } \ell_1, \mathsf{balance}_1 \;=\; \mathbf{acquire}(\ell_1) \textbf{ in} \\
&\quad \textbf{let } \ell_2, \mathsf{balance}_2 \;=\; \mathbf{acquire}(\ell_1) \textbf{ in} \\
&\quad \textbf{if } \mathsf{balance}_1 \geq \mathsf{amount} \textbf{ then} \\
&\qquad (\mathbf{release}(\ell_1, \mathsf{balance}_1 - \mathsf{amount}), \mathbf{release}(\ell_2, \mathsf{balance}_2 + \mathsf{amount})) \\
&\quad \textbf{else} \\
&\qquad (\mathbf{release}(\ell_1, \mathsf{balance}_1), \mathbf{release}(\ell_2, \mathsf{balance}_2))
\end{aligned}$$

Note that we did not have to keep track of any lock orders, or had to do any other analysis to show that this does not deadlock, regardless of what the rest of the program does. In Section 6 we introduce lock groups, which allow us to extend this example to multiple bank threads sharing multiple locks, still ensuring deadlock and memory leak freedom.

2.3.5 Shared mutable recursive data structures. We can define a recursive type tree where each node is protected by a lock and stores a value of type τ:

$$\mathsf{tree} \triangleq \mathbf{Lock}\langle 1 + \mathsf{tree} \times \tau \times \mathsf{tree}\,{}^{1}_{0}\rangle$$

These trees own their children. In order to operate over such trees conurrently, we define the type tree' of client references to trees:

$$\mathsf{tree}' \triangleq \mathbf{Lock}\langle 1 + \mathsf{tree} \times \tau \times \mathsf{tree}\,{}^{0}_{0}\rangle$$

The main thread can now allocate a tree, and share multiple client references of type tree' with child threads. Using a client reference we can not only modify the root, but we can also traverse the tree. For instance, to try and obtain a client reference to the left child (if any), we acquire the lock, create a client reference to the left child (using **fork**), and release the lock:

$$\begin{aligned}
&\mathsf{left} : \mathsf{tree}' \multimap 1 + \mathsf{tree}' \\
&\mathsf{left}(\ell) \triangleq \\
&\quad \textbf{let } \ell, t \;=\; \mathbf{acquire}(\ell) \textbf{ in} \\
&\quad \textbf{match } t \textbf{ with} \\
&\quad \mathbf{in_L}() \Rightarrow \mathbf{release}(\ell, \mathbf{in_L}()); \mathbf{in_L}() \\
&\quad \mathbf{in_R}(\ell_1, x, \ell_2) \Rightarrow \mathbf{in_R}(\mathbf{fork}(\ell_1, \lambda \ell_1.\ \mathbf{release}(\ell, \mathbf{in_R}(\ell_1, x, \ell_2)))) \\
&\quad \textbf{end}
\end{aligned}$$

Note that **fork** operates as an administrative device here; when we acquire the lock ℓ we obtain owning references ℓ_1, ℓ_2 to the children, and fork allows us to obtain a client reference for ℓ_1 while putting the owning references back in the lock ℓ. One would not actually fork a thread in a real implementation.

Because we immediately release the lock after obtaining a child reference, we can have multiple threads operate on different parts on the tree concurrently, while guaranteeing leak and deadlock freedom.

2.3.6 Client-server. Our language also comes equipped with linear channels for message-passing concurrency that can be session-typed, thanks to our encoding detailed in Section 3.1. Using locks, we can share a channel endpoint among multiple participants, which allows us to implement a client-server pattern, as is possible in the deadlock-free fragment of manifest sharing [Balzer and

Pfenning 2017].

$$\begin{aligned}
&\textbf{let } c \;=\; (\cdots \text{create new server channel} \cdots) \textbf{ in}\\
&\textbf{let } \ell \;=\; \textbf{release}(\textbf{new}(), c) \textbf{ in}\\
&\textbf{let } \ell \;=\; \textbf{fork}(\ell, \lambda \ell.\ \cdots) \textbf{ in}\\
&\textbf{let } \ell \;=\; \textbf{fork}(\ell, \lambda \ell.\ \cdots) \textbf{ in}\\
&\cdots\\
&\textbf{let } c \;=\; \textbf{wait}(\ell)
\end{aligned}$$

Each client can temporarily take the lock, which allows it to interact with the server. As in [Balzer et al. 2019], typing ensures that a lock must be released to the same protocol state at which it was previously acquired, ensuring type safety.

2.3.7 Locks over channels. The preceding example involves putting channels in locks, but we can also send locks over channels. For instance, one can send an open lock acting like a future/promise to another thread, so that the other thread gets the obligation to fulfill the promise by storing a value in the lock.

2.3.8 Encoding session-typed channels. In Section 3.1 we show that we can implement session-typed channels using our locks.

2.4 Sharing multiple locks with lock orders

The simple system illustrated above is restricted to sharing only one lock at each fork. We lift this restriction in Section 6 by introducing *lock groups*. Lock groups consist of multiple locks, and one is allowed to share an entire lock group at each fork. In turn, we must introduce another mechanism to ensure leak and deadlock freedom within a lock group. We do this by imposing a lock order on the locks of a lock group, and requiring that the locks are acquired in increasing order. A similar condition takes care that there is no deadlock between several **wait**s or between **wait** and **acquire**. In Section 6.1 we provide examples of the use of lock orders. In particular, we can handle a version of Dijkstra's dining philosophers problem with a *dynamic* number of participants dependent on a run-time variable n.

Importantly, deadlock freedom between lock groups is taken care of by the sharing topology, so one is always free to acquire locks from different lock groups, and do transactions between different lock groups in that manner. This makes lock groups more *compositional* than standard global lock orders that require a global order on the entire system whenever multiple locks are acquired.

3 THE λ_{lock} LANGUAGE

We give a formal description of λ_{lock}'s syntax, type system, and operational semantics. The base of λ_{lock} is a linear λ-calculus, extended with unrestricted types (whose values can be freely duplicated, dropped, and deallocated) and recursive types:

$$\tau \in \textbf{Type} \triangleq \mathbf{0} \mid \mathbf{1} \mid \tau + \tau \mid \tau \times \tau \mid \tau \multimap \tau \mid \tau \to \tau \mid \textbf{Lock}\langle \tau \,{}^{a}_{b}\rangle \mid \mu x.\tau \mid x$$

We distinguish linear functions $\tau_1 \multimap \tau_2$ from unrestricted functions $\tau_1 \to \tau_2$. Unrestricted functions can be freely duplicated and discarded, and hence can only capture unrestricted variables. Linear functions, on the other hand, must be treated linearly, and hence can close over both linear and unrestricted variables. Rather than distinguishing sums and products into linear and unrestricted, we consider sums and products to be unrestricted if their components are. Similarly, we consider recursive types to be unrestricted if their coinductive unfoldings are (see Section 7). The empty type $\mathbf{0}$ and unit type $\mathbf{1}$ are always unrestricted. The lock type $\textbf{Lock}\langle \tau \,{}^{a}_{b}\rangle$ is always linear, regardless of whether τ is.

$$\frac{\Gamma\ \mathsf{unr}}{\Gamma \vdash \mathbf{new}() : \mathbf{Lock}\langle \tau\ {}^{1}_{1} \rangle} \qquad \frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau\ {}^{0}_{0} \rangle}{\Gamma \vdash \mathbf{drop}(e) : \mathbf{1}} \qquad \frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau\ {}^{1}_{0} \rangle}{\Gamma \vdash \mathbf{wait}(e) : \tau}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{Lock}\langle \tau\ {}^{a_1+a_2}_{b_1+b_2} \rangle \qquad \Gamma_2 \vdash e_2 : \mathbf{Lock}\langle \tau\ {}^{a_2}_{b_2} \rangle \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}(e_1, e_2) : \mathbf{Lock}\langle \tau\ {}^{a_1}_{b_1} \rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{Lock}\langle \tau\ {}^{a}_{0} \rangle}{\Gamma \vdash \mathbf{acquire}(e) : \mathbf{Lock}\langle \tau\ {}^{a}_{1} \rangle \times \tau} \qquad \frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{Lock}\langle \tau\ {}^{a}_{1} \rangle \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma \vdash \mathbf{release}(e_1, e_2) : \mathbf{Lock}\langle \tau\ {}^{a}_{0} \rangle}$$

Fig. 1. λ_{lock}'s lock typing rules.

Our language λ_{lock} has the following syntax:

$$\begin{aligned} e \in \mathit{Expr} ::= {} & x \mid () \mid (e, e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\ e \mid e\ e \mid \mathbf{let}\ (x_1, x_2)\ =\ e\ \mathbf{in}\ e \mid \\ & \mathbf{match}\ e\ \mathbf{with}\ \bot\ \mathbf{end} \mid \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} \mid \\ & \mathbf{new}() \mid \mathbf{fork}(e, e) \mid \mathbf{acquire}(e) \mid \mathbf{release}(e, e) \mid \mathbf{drop}(e) \mid \mathbf{wait}(e) \end{aligned}$$

The typing rules for the lock operations can be found in Figure 1, and the typing rules for the base language can be found in Figure 2. We use the judgments $\Gamma\ \mathsf{unr}$ and $\Gamma \equiv \Gamma_1 \cdot \Gamma_2$ to handle linear and unrestricted types: $\Gamma\ \mathsf{unr}$ means that all types in Γ are unrestricted, and $\Gamma \equiv \Gamma_1 \cdot \Gamma_2$ splits up Γ into Γ_1 and Γ_2 disjointly for variables of linear type, while allowing variables of unrestricted type to be shared by both Γ_1 and Γ_2. We do not include a constructor for recursive functions, because recursive functions can already be encoded in terms of recursive types, using the Y-combinator.[1]

The rules for the operational semantics can be found in Figure 3. We use a small step operational semantics built up in two layers. The first layer defines values, evaluation contexts, and reductions for pure expressions. The values are standard for λ-calculus, except for $\langle k \rangle$, which indicates a reference/pointer to a lock identified by the number k.

The second layer operates on a *configuration*, which is a collection of threads and locks, each identified with a natural number. A thread $\mathsf{Thread}(e)$ comprises the expression e that it executes, and a lock $\mathsf{Lock}(\mathrm{refcnt}, \mathbf{None} \mid \mathbf{Some}(v))$ comprises a reference count refcnt (*i.e.,* the number of client references) and either **None**, indicating that the lock has been acquired and currently contains no value, or $\mathbf{Some}(v)$, indicating that the lock is currently closed and is holding the value v.

The stepping rules for the configuration are as follows, as labeled in Figure 3.

pure Perform a pure reduction in an evaluation context.

new Allocate a new lock at a fresh position k, and return a reference $\langle k \rangle$ to the thread.

fork Fork off a new thread, while duplicating the reference to lock k, passing $\langle k \rangle$ back to the main thread, as well as to the new child thread.

acquire If the lock currently contains $\mathbf{Some}(v)$, then the acquire can proceed, and returns the value to the thread and puts **None** in the lock.

release Does the opposite: replaces **None** in the lock with $\mathbf{Some}(v)$, where v is the value provided to the release operation.

drop Deletes a reference to the lock, decrementing its reference count.

wait When the reference count is 0 and there is a $\mathbf{Some}(v)$ in the lock, the operation can proceed and removes the lock from the configuration, while giving the value to the thread.

[1] Of course, for efficiency of an implementation one wants direct support for recursion.

$$\frac{\Gamma \text{ unr}}{\Gamma, x{:}\tau \vdash x : \tau} \qquad \frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \text{ unr} \qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \text{ unr}}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau_3}{\Gamma \vdash \mathbf{let}\ x_1, x_2\ =\ e_1\ \mathbf{in}\ e_2 : \tau_3} \qquad \frac{\Gamma \text{ unr} \qquad \Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \bot\ \mathbf{end} : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{in_L}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{in_R}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e : \tau_1 + \tau_2 \qquad \Gamma_2, x_1{:}\tau_1 \vdash e_1 : \tau' \qquad \Gamma_2, x_2{:}\tau_2 \vdash e_2 : \tau'}{\Gamma \vdash \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} : \tau'}$$

Fig. 2. λ_{lock}'s base linear λ-calculus with sums & products and linear & unrestricted functions.

exit When a thread has terminated with a unit value, we remove the thread from the configuration.

frame Closes the set of preceding rules under disjoint union with a remaining configuration. This allows the preceding rules to take place within a large configuration.

3.1 Encoding session-typed channels

One can implement session-typed channels using our locks. Consider basic session types [Honda 1993; Lindley and Morris 2015; Wadler 2012]:

$$s \in \mathit{Session} ::= !\tau.s \mid ?\tau.s \mid s \,\&\, s \mid s \oplus s \mid \mathsf{End}_! \mid \mathsf{End}_?$$

We can implement the usual channel operations as follows:

$$\begin{aligned}
\mathbf{fork_C}(f) &\triangleq \mathbf{fork}(\mathbf{new}(), f)\\
\mathbf{send_C}(c, v) &\triangleq \mathbf{fork}(\mathbf{new}(), \lambda c'.\, \mathbf{drop}(\mathbf{release}(c, (c', v))))\\
\mathbf{receive_C}(c) &\triangleq \mathbf{wait}(c)\\
\mathbf{tell_L}(c) &\triangleq \mathbf{fork}(\mathbf{new}(), \lambda c'.\, \mathbf{drop}(\mathbf{release}(c, \mathbf{in_L}(c'))))\\
\mathbf{tell_R}(c) &\triangleq \mathbf{fork}(\mathbf{new}(), \lambda c'.\, \mathbf{drop}(\mathbf{release}(c, \mathbf{in_R}(c'))))\\
\mathbf{ask}(c) &\triangleq \mathbf{wait}(c)\\
\mathbf{close_C}(c) &\triangleq \mathbf{drop}(\mathbf{release}(c, ()))\\
\mathbf{wait_C}(c) &\triangleq \mathbf{wait}(c)
\end{aligned}$$

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid (v, v) \mid \mathbf{in_L}(v) \mid \mathbf{in_R}(v) \mid \lambda x.\ e \mid \langle k \rangle \\
K \in \mathit{Ctx} &::= \Box \mid (K, e) \mid (v, K) \mid \mathbf{in_L}(K) \mid \mathbf{in_R}(K) \mid K\ e \mid v\ K \mid \mathbf{let}\ x_1, x_2\ =\ K\ \mathbf{in}\ e \\
&\quad \mid \mathbf{match}\ K\ \mathbf{with}\ \bot\ \mathbf{end} \mid \mathbf{match}\ K\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} \\
&\quad \mid \mathbf{new}(K) \mid \mathbf{fork}(K, e) \mid \mathbf{fork}(v, K) \mid \mathbf{acquire}(K) \mid \mathbf{release}(K, e) \\
&\quad \mid \mathbf{release}(v, K) \mid \mathbf{drop}(K) \mid \mathbf{wait}(K)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{match}\ \mathbf{in_L}(v)\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1 \mid \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} &\rightsquigarrow_{\text{pure}} e_1[v/x_1] \\
\mathbf{match}\ \mathbf{in_R}(v)\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1 \mid \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} &\rightsquigarrow_{\text{pure}} e_2[v/x_2] \\
\mathbf{let}\ x_1, x_2\ =\ (v_1, v_2)\ \mathbf{in}\ e &\rightsquigarrow_{\text{pure}} e[v_1/x_1][v_2/x_2] \\
(\lambda x.\ e)\ v &\rightsquigarrow_{\text{pure}} e[v/x]
\end{aligned}
$$

$$\rho \in \mathsf{Cfg} \triangleq \mathbb{N} \xrightarrow{\text{fin}} \mathsf{Thread}(e) \mid \mathsf{Lock}(\mathsf{refcnt}, \mathbf{None} \mid \mathbf{Some}(v))$$

$$\{n \mapsto \mathsf{Thread}(K[\ e_1\ ])\} \overset{n}{\rightsquigarrow} \{n \mapsto \mathsf{Thread}(K[\ e_2\ ])\} \quad \text{if } e_1 \rightsquigarrow_{\text{pure}} e_2 \qquad \text{(pure)}$$

$$\{n \mapsto \mathsf{Thread}(K[\ \mathbf{new}()\ ])\} \overset{n}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \langle k \rangle\ ]) \\ k \mapsto \mathsf{Lock}(0, \mathbf{None}) \end{Bmatrix} \qquad \text{(new)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \mathbf{fork}(\langle k \rangle, v)\ ]) \\ k \mapsto \mathsf{Lock}(\mathsf{refcnt}, x) \end{Bmatrix} \overset{k}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \langle k \rangle\ ]) \\ m \mapsto \mathsf{Thread}(v\ \langle k \rangle) \\ k \mapsto \mathsf{Lock}(1 + \mathsf{refcnt}, x) \end{Bmatrix} \qquad \text{(fork)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \mathbf{acquire}(\langle k \rangle)\ ]) \\ k \mapsto \mathsf{Lock}(\mathsf{refcnt}, \mathbf{Some}(v)) \end{Bmatrix} \overset{k}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ (\langle k \rangle, v)\ ]) \\ k \mapsto \mathsf{Lock}(\mathsf{refcnt}, \mathbf{None}) \end{Bmatrix} \qquad \text{(acquire)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \mathbf{release}(\langle k \rangle, v)\ ]) \\ k \mapsto \mathsf{Lock}(\mathsf{refcnt}, \mathbf{None}) \end{Bmatrix} \overset{k}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \langle k \rangle\ ]) \\ k \mapsto \mathsf{Lock}(\mathsf{refcnt}, \mathbf{Some}(v)) \end{Bmatrix} \qquad \text{(release)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \mathbf{drop}(\langle k \rangle)\ ]) \\ k \mapsto \mathsf{Lock}(1 + \mathsf{refcnt}, x) \end{Bmatrix} \overset{k}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ ()\ ]) \\ k \mapsto \mathsf{Lock}(\mathsf{refcnt}, x) \end{Bmatrix} \qquad \text{(drop)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \mathbf{wait}(\langle k \rangle)\ ]) \\ k \mapsto \mathsf{Lock}(0, \mathbf{Some}(v)) \end{Bmatrix} \overset{k}{\rightsquigarrow} \{n \mapsto \mathsf{Thread}(K[\ v\ ])\} \qquad \text{(wait)}$$

$$\{n \mapsto \mathsf{Thread}(())\} \overset{n}{\rightsquigarrow} \{\} \qquad \text{(exit)}$$

$$\rho_1 \uplus \rho' \overset{i}{\rightsquigarrow} \rho_2 \uplus \rho' \quad \text{if } \rho_1 \overset{i}{\rightsquigarrow} \rho_2 \quad (\uplus \text{ is disjoint union}) \qquad \text{(frame)}$$

Fig. 3. λ_{lock}'s operational semantics.

Of course, implementing channels this way is inefficient, because a tiny thread is forked every time we send a message. Thus, it is still worth having native channels, or a compiler that supports a version of fork that does not actually spawn a new thread, but runs the body immediately (though care must be taken not to introduce deadlocks). To type these operations, we use the following

definition of session types in terms of locks, where $\overline{s}$ is the dual of s:

$$!\tau.s \triangleq \mathbf{Lock}\langle \overline{s} \times \tau \,{}^{0}_{1}\rangle$$
$$?\tau.s \triangleq \mathbf{Lock}\langle s \times \tau \,{}^{1}_{0}\rangle$$
$$s_1 \,\&\, s_2 \triangleq \mathbf{Lock}\langle \overline{s_1} + \overline{s_2} \,{}^{0}_{1}\rangle$$
$$s_1 \oplus s_2 \triangleq \mathbf{Lock}\langle s_1 + s_2 \,{}^{1}_{0}\rangle$$
$$\mathsf{End}_! \triangleq \mathbf{Lock}\langle \mathbf{1} \,{}^{0}_{1}\rangle$$
$$\mathsf{End}_? \triangleq \mathbf{Lock}\langle \mathbf{1} \,{}^{1}_{0}\rangle$$

This encoding resembles the encodings of [Dardha et al. 2012, 2017; Kobayashi 2002b]. After encoding session types this way, we can type check the channel operations with the standard session typing rules:

$$\mathbf{fork}_{\mathrm{C}} : (s \multimap \mathbf{1}) \multimap \overline{s}$$
$$\mathbf{close}_{\mathrm{C}} : \mathsf{End}_! \multimap \mathbf{1}$$
$$\mathbf{wait}_{\mathrm{C}} : \mathsf{End}_? \multimap \mathbf{1}$$
$$\mathbf{send}_{\mathrm{C}} : !\tau.s \times \tau \multimap s$$
$$\mathbf{receive}_{\mathrm{C}} : ?\tau.s \multimap s \times \tau$$
$$\mathbf{tell}_{\mathrm{L}} : s_1 \,\&\, s_2 \multimap s_1$$
$$\mathbf{tell}_{\mathrm{R}} : s_1 \,\&\, s_2 \multimap s_2$$
$$\mathbf{ask} : s_1 \oplus s_2 \multimap s_1 + s_2$$

Because we can encode these session-typed channels in our deadlock and memory leak free language, this automatically shows that these session-typed channels are deadlock and memory leak free. Note that the encoding of session types relies in an essential way on higher-order locks.

4 THE DEADLOCK AND LEAK FREEDOM THEOREMS

Our goal was to make λ_{lock} deadlock and memory leak free. We will now make this more precise. Firstly, let us look at how the usual notion of type safety can be adapted to a language with blocking constructs. Type safety for a single threaded language says that if we start with a well-typed program, then the execution of the program doesn't get stuck until we terminate with a value. If we have multiple threads, we could say that this has to hold *for every thread*, but if we have blocking constructs this is clearly not true: a thread can temporarily get stuck while blocking. We therefore modify the notion of type safety to say that each thread can always make progress, except if the thread is legitimately blocked, *i.e.,* blocked on an operation that is supposed to block, like **acquire**.

This, of course, is not a strong enough property for our purposes. To rule out deadlocks, we want to say that even if *some* threads are blocked, there is always *at least one* thread that can make progress. Furthermore, we wish to say that if all threads terminate, then all memory has been deallocated. Because of the way we have set up our operational semantics, we can formulate this simply as saying: if the configuration cannot step, then it must be empty.

Let us consider the formal statement of global progress:

Theorem 4.1 (Global progress).
If $\emptyset \vdash e : \mathbf{1}$, *and* $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho$, *then either* $\rho = \{\}$ *or* $\exists \rho'.\, \rho \rightsquigarrow \rho'$.

Global progress rules out whole-program deadlocks, and it ensures that all locks have been deallocated when the program terminates. However, it does not guarantee anything as long as

there is still a single thread that can step. Thus it only guarantees a weak form of deadlock freedom, and it only guarantees memory leak freedom when the program terminates, not during execution.

To formulate stronger forms of deadlock and leak freedom, we take an approach similar to the approaches previously taken for session types [Jacobs et al. 2022a]. Namely, we define the relation $i \ \mathsf{waiting}_\rho \ j$, which says that $i \in \mathsf{dom}(\rho)$ is waiting for $j \in \mathsf{dom}(\rho)$. Intuitively, in the graph of connections between objects in the configuration (*i.e.,* between threads and locks, and between locks and locks), we give each such connection a waiting direction, so that either $i \ \mathsf{waiting}_\rho \ j$, or $j \ \mathsf{waiting}_\rho \ i$. We define this relation such that if i is a thread, and currently about to execute a lock operation, then $i \ \mathsf{waiting}_\rho \ j$. Furthermore, in all other cases, we say that $j \ \mathsf{waiting}_\rho \ i$, if there is some reference from i to j or from j to i.

Consider our operational semantics stepping rule $\rho \overset{i}{\rightsquigarrow} \rho'$: this step relation is annotated with a number i, indicating which object in the configuration we consider responsible for the step. The waiting relation sets up a blame game with respect to this step relation: whenever we ask some object i why the configuration isn't making progress, i should either respond that it can make the configuration step (*i.e.,* $\exists \rho', \rho \overset{i}{\rightsquigarrow} \rho'$), or i should blame somebody else, by showing $\exists j, i \ \mathsf{waiting}_\rho \ j$.

We can then continue to ask j why the configuration isn't making progress, and so on. Since we maintain the invariant that the graph of connections is acyclic, it is not possible that the blame game loops back to the original i in a cycle, since then we'd either have a cycle in the reference structure. Furthermore, the blame game cannot revisit i via the same edge that was used to leave i either, since then we'd have $i \ \mathsf{waiting}_\rho \ j$ and $j \ \mathsf{waiting}_\rho \ i$ for some j, which is impossible due to the way we've defined $\mathsf{waiting}_\rho$. Therefore we conclude that the blame game must eventually terminate at some $j \in \mathsf{dom}(\rho)$ who shows that the configuration can step.

Importantly, this gives us a stronger theorem, namely that if we start at any $i \in \mathsf{dom}(\rho)$, there is some j *transitively connected to* i *via waiting dependencies*, such that j can make the configuration step. This will rule out that a subset of the configuration has been leaked or deadlocked, because in that case there would be no such transitive path to a thread that can step.

In contrast to Jacobs et al. [2022a], we define these notions more generically, so that we only need to prove *one* language specific theorem, from which all the other properties that we wish to establish follow generically, without further dependence on the particular language under consideration.

Let us now look at this in more formal detail. A language specific piece of information we need is the relation e blocked k, which says that expression e is blocked on the object identified by k. Note that unlike the waiting relation, this relation does not depend on the configuration; whether an expression e is blocked can be determined from the expression itself:

Definition 4.1. We have e blocked k if e is of the form $K[\, e_0 \,]$ for some evaluation context K, and e_0 is one of $\mathbf{fork}(\langle k \rangle, v)$, $\mathbf{acquire}(\langle k \rangle)$, $\mathbf{release}(\langle k \rangle, v)$, $\mathbf{drop}(\langle k \rangle)$, $\mathbf{wait}(\langle k \rangle)$, for some value v.

Note that we formally include all the lock operations in the blocked relation, even the ones that are conventionally not thought of as blocking. The reason we do this is because we consider the locks to be responsible for the step operations involving the lock, and not the thread, as can be seen from the annotations i on the step relation $\rho \overset{i}{\rightsquigarrow} \rho'$ in the operational semantics (Figure 3). This streamlines the formal statements because they become more uniform.

Secondly, we need to be able to determine all the outgoing references for an object in the configuration:

Definition 4.2. We have the function $\mathsf{refs}_\rho(i) \subseteq \mathsf{dom}(\rho)$, which gives the set of all references $\langle k \rangle$ stored inside $\rho(i)$. We omit the formal definition here, as this can be defined using a straightforward recursion on expressions and values.

This allows us to define the waiting relation:

Definition 4.3. We have i $\mathsf{waiting}_\rho$ j if either:

(1) $\rho(i) = \mathsf{Thread}(e)$ and e blocked j.
(2) $i \in \mathsf{refs}_\rho(j)$ and not $\rho(j) = \mathsf{Thread}(e)$ with e blocked i.

That is, threads are waiting for the objects they are blocked on, and if an object has an incoming reference and this reference is not from a thread blocked on that object, then the object is considered to be waiting for the source of the incoming reference. Specifically, if a thread has a reference to a lock, and the thread is not currently about to execute an operation with this lock, then the lock is said to be waiting for the thread. Similarly, if a lock holds a reference to a lock, then the second lock is considered to be waiting for the first.

Using the waiting relation notion, we can formally define what a partial deadlock/leak is. Intuitively, a partial deadlock is a subset of the objects in the configuration, none of which can step, such that if an object in the deadlock is waiting, then it must be waiting for another object in the deadlock.

Definition 4.4 (Partial deadlock/leak)**.** Given a configuration ρ, a non-empty subset $S \subseteq \mathsf{dom}(\rho)$ is in a partial deadlock/leak if these two conditions hold:

(1) No $i \in S$ can step, *i.e.,* for all $i \in S$, $\neg\exists\rho'.\ \rho \overset{i}{\rightsquigarrow} \rho'$
(2) If $i \in S$ and i $\mathsf{waiting}_\rho$ j then $j \in S$

This notion also incorporates memory leaks: if there is some lock that is not referenced by a thread or other lock, then the singleton set of that lock is a partial deadlock/leak. Furthermore, if we have a set of locks that all reference only each other circularly, then this is considered to be a partial deadlock/leak. Similarly, a single thread that is not synchronizing on a lock, is considered to be in a singleton deadlock if it cannot step.

Definition 4.5 (Partial deadlock/leak freedom)**.** A configuration ρ is deadlock/leak free if no $S \subseteq \mathsf{dom}(\rho)$ is in a partial deadlock/leak.

We can reformulate this to look more like the standard notion of memory leak freedom, namely reachability:

Definition 4.6 (Reachability)**.** We inductively define the threads and locks that are *reachable* in ρ: $j_0 \in N$ is reachable in ρ if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geq 0$) such that j_0 $\mathsf{waiting}_\rho$ j_1, and j_1 $\mathsf{waiting}_\rho$ j_2, ..., and j_{k-1} $\mathsf{waiting}_\rho$ j_k, and finally j_k can step in ρ, *i.e.,* $\exists\rho'.\ \rho \overset{j_k}{\rightsquigarrow} \rho'$.

Intuitively, an element of the configuration is reachable if it can step, or if it has a transitive waiting dependency on some other element that can step. This notion is stronger than the usual notion of reachability, which considers objects to be reachable even if they are only reachable from threads that are blocked.

Definition 4.7 (Full reachability)**.** A configuration ρ is *fully reachable* if all $i \in \mathsf{dom}(\rho)$ are reachable in ρ.

As in [Jacobs et al. 2022a], our strengthened formulations of deadlock freedom and full reachability are equivalent for λ_{lock}:

Theorem 4.2. *A configuration ρ is deadlock/leak free if and only if it is fully reachable.*

In contrast to [Jacobs et al. 2022a], we have carefully set up our definitions so that this theorem holds fully generically, *i.e.,* independent of any language specific properties.

These notions also imply global progress and type safety:

Definition 4.8. A configuration ρ satisfies global progress if $\rho = \emptyset$ or $\exists \rho', i.\ \rho \overset{i}{\rightsquigarrow} \rho'$.

Definition 4.9. A configuration ρ is safe if for all $i \in \mathsf{dom}(\rho)$, $\exists \rho', i.\ \rho \overset{i}{\rightsquigarrow} \rho'$, or $\exists j.\ i\ \mathsf{waiting}_\rho\ j$.

That is, global progress holds if there is any element of the configuration that can step, and safety holds if all elements of the configuration can either step, or are legally blocked (*i.e.,* waiting for someone else).

Theorem 4.3. *If a configuration ρ is fully reachable, then ρ has the progress and safety properties.*

We thus only need to prove one language specific theorem, namely that all configurations that arise from well-typed programs are fully reachable:

Theorem 4.4. *If $\emptyset \vdash e : \mathbf{1}$ and $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho'$, then ρ' is fully reachable.*

Once we have this theorem, the other theorems follow.
In the next section (Section 5) we give a high-level overview of how the theorem is proved.

5 AN INTUITIVE DESCRIPTION OF THE PROOFS

In this section we give a high-level overview of the proof of Theorem 4.4. We keep the discussion high-level and intuitive because the full details are in the mechanized proofs (Section 7).

Recall that Theorem 4.4 says that if we start with a well-typed program, then every object in the configuration always remains reachable (Definition 4.6). In order to show this, we will structure the proof in the style of progress and preservation: we first define an invariant on the configuration, showing that the invariant is preserved by the operational semantics (analogous to preservation), and then show that configurations that satisfy the invariant are fully reachable (analogous to progress). Thus, our first task is to come up with a suitable invariant.

As we have seen in Section 2, our invariant must ensure that the sharing topology in the configuration is acyclic. That is, if one considers the graph where the threads and locks are vertices, and there is an (undirected) edge between two vertices if there is a reference between them (in either direction), then this graph shall remain acyclic.

Another aspect of our invariant is well-typedness: we must ensure that the expressions of every thread, and the values in every lock, are well-typed. Furthermore, if there are lock references $\langle k \rangle$ in expressions or values, then the type assigned to these must be consistent with the type of values actually stored in the lock.

However, the type of lock references is, in general, $\langle k \rangle : \mathbf{Lock}\langle \tau\, {}^{a}_{b} \rangle$. The invariant must also account for the consistency of the a and b of the different references to the same lock. We require the following conditions for a lock $\{k \mapsto \mathsf{Lock}(\mathsf{refcnt}, v)\}$ in the configuration:

- Out of all the references $\langle k \rangle$ appearing in the configuration, precisely one has $a = 1$.
- Out of all the references $\langle k \rangle$ appearing in the configuration, at most one has $b = 1$. Furthermore, if $v = \mathbf{Some}(v')$, we must have $b = 0$ for all references, and if $v = \mathbf{None}$, we must have precisely one reference with $b = 1$.
- The number of references with $a = 0$ must be consistent with refcnt.

We capture the acyclicity condition, the well-typedness condition, and the lock conditions in a predicate $\mathsf{inv}(\rho)$ on configurations, which states that the configuration ρ satisfies these conditions. Our invariant is that this predicate holds throughout execution. Formally, we have to show:

Theorem 5.1 (preservation). *If $\rho \overset{i}{\rightsquigarrow} \rho'$ then $\mathsf{inv}(\rho) \implies \mathsf{inv}(\rho')$*

The proof of this theorem involves careful mathematical reasoning about the sharing topology: we must show that each lock operation, although it may modify the structure of the graph, ensures

that if the graph of ρ was acyclic, then the graph of ρ' is also acyclic, provided the program we are executing is well-typed in the linear type system.

Secondly, we must ensure that all operations leave all the expressions and values in the configuration well-typed, and maintains the correctness conditions regarding the references to each lock.

Having done this, it remains to show that if a configuration satisfies our invariant, then every object in the configuration is reachable:

Theorem 5.2 (reachability). *If* $\mathsf{inv}(\rho)$*, then* ρ *is fully reachable.*

Recall Definition 4.6 of reachability: an object j_0 in the configuration is reachable, if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geq 0$) such that j_0 $\mathsf{waiting}_\rho$ j_1, and j_1 $\mathsf{waiting}_\rho$ j_2, ..., and j_{k-1} $\mathsf{waiting}_\rho$ j_k, and finally j_k can step in ρ, *i.e.,* $\exists \rho'.\ \rho \overset{j_k}{\rightsquigarrow} \rho'$. Thus, proving Theorem 5.2 amounts to constructing such a sequence and showing that the final element in the sequence can step. To construct this sequence, we must rely on the acyclicity of the sharing topology in an essential way.

We do this by formulating our strategy for constructing such a sequence with respect to that graph: we start at the vertex j_0, check whether j_0 can itself step, and if not, show that there must exist some j_1 such that j_0 $\mathsf{waiting}_\rho$ j_1. We then repeat this process iteratively.

There is a danger that this process does not terminate (*i.e.,* goes in a cycle, since the configuration is finite), but by being careful we can avoid this danger:

(1) We make sure, that if we step from j_i to j_{i+1}, that there is an edge between j_i and j_{i+1} in the acyclic graph.
(2) We make sure that if we just stepped from j_i to j_{i+1}, we will not immediately step back from j_{i+1} to j_i.

These two conditions together ensure that our stepping process is well-founded: if we step along edges in an acyclic graph, and never turn around and step back, then we must eventually arrive at some leaf vertex with only one adjacent edge, from which we just came, and we are then forced to stop.

Thus, in order to prove Theorem 5.2, it is sufficient to come up with a stepping strategy, and show that it satisfies these two conditions. This strategy, roughly speaking, works as follows:

- If we are currently at a thread i with $\rho(i) = \mathsf{Thread}(e)$, then by well-typedness of the expression e we can show that the thread can either take a step, or it is currently attempting to execute a lock operation on some lock $\langle k \rangle$. In the former case, we are done. In the latter case, we have i $\mathsf{waiting}_\rho$ k, so we step to vertex k, and continue building our sequence of transitive waiting dependencies from there.
- If we are currently at a lock k with $\rho(k) = \mathsf{Lock}(\mathsf{refcnt}, v)$, we can show, from the invariant we maintain about locks, that we are in one of the following situations:

(1) We have $v = \mathbf{None}$ and there is an incoming reference $\langle k \rangle$ from some $j \in \mathsf{dom}(\rho)$ with $\langle k \rangle : \mathbf{Lock}\langle \tau\, {}^{a}_{1} \rangle$, *i.e.,* an opened reference.
(2) We have $v = \mathbf{Some}(v')$ and $\mathsf{refcnt} = 0$, and there an incoming reference $\langle k \rangle$ from some $j \in \mathsf{dom}(\rho)$ with $\langle k \rangle : \mathbf{Lock}\langle \tau\, {}^{1}_{0} \rangle$, *i.e.,* a closed owner reference.
(3) We have $v = \mathbf{Some}(v')$ and $\mathsf{refcnt} \neq 0$, and there an incoming reference $\langle k \rangle$ from some $j \in \mathsf{dom}(\rho)$ with $\langle k \rangle : \mathbf{Lock}\langle \tau\, {}^{0}_{0} \rangle$, *i.e.,* a closed client reference.

In each case, if j is a lock, then we are immediately done because we can show i $\mathsf{waiting}_\rho$ j and step to j. If j is a thread, then we have two cases:

– The thread is waiting for us, *i.e.,* trying to do a lock operation with $\langle k \rangle$. In this case, the above information guarantees that this lock operation can proceed:

(1) In the first case with $v = \mathbf{None}$, we know that the only lock operations that are allowed by the type $\langle k \rangle : \mathbf{Lock}\langle \tau\, {}^{a}_{1} \rangle$ are **release** and **fork**, both of which can proceed. In particular, since the lock is open, the **wait** operation, which could block, is not permitted.
(2) In the second case with $v = \mathbf{Some}(v')$ and refcnt $= 0$, we have a closed owner reference, so the only potentially blocking operation that is permitted is **wait**, which can proceed since refcnt $= 0$.
(3) In the third case with $v = \mathbf{Some}(v')$ and refcnt $\neq 0$, we have a closed client reference, so none of the operations permitted are blocking.

– The thread is not waiting for us, *i.e.,* we are waiting for the thread. So we step to the thread, and continue building our sequence of transitive waiting dependencies from there.

This concludes the sketch of the proofs of Theorem 5.1 and Theorem 5.2, which together can be used to prove our main Theorem 4.4, from which all the other theorems in Section 4 follow. Although the description of the proofs here omit many details, particularly with respect to the preservation of acyclicity of the sharing topology, the description is nevertheless faithful to how the mechanized Coq proof is done.

6 THE $\lambda_{\mathbf{lock{+}{+}}}$ LANGUAGE: SHARING MULTIPLE LOCKS WITH LOCK GROUPS

The language λ_{lock} we have seen so far only allows us to share *one* lock with a child thread when we fork. To alleviate this restriction we now develop $\lambda_{\text{lock++}}$, which allows us to share more than one lock with a child thread. This allows us to handle locally cyclic connections.

The mechanism by which $\lambda_{\text{lock++}}$ allows this is *lock groups*. The locks of λ_{lock} store one pair $(\text{refcnt}, \mathbf{None} \mid \mathbf{Some}(v))$ of a reference count and an optional value, whereas the lock groups of $\lambda_{\text{lock++}}$ store a collection of such pairs. Whereas the type of locks in λ_{lock} is $\mathbf{Lock}\langle \tau\, {}^{a}_{b} \rangle$, the type of a lock group in $\lambda_{\text{lock++}}$ is:

$$\mathbf{Lock}_{\mathrm{G}}\langle \tau_1\, {}^{a_1}_{b_1}, \ldots, \tau_n\, {}^{a_n}_{b_n} \rangle$$

That is, we generalize the data within the brackets from one item to n items, but each item still consists of a triple τ^a_b where τ indicates the type of that lock, a indicates whether we are the owner of that lock, and b indicates whether we have currently acquired that lock.

We are allowed to freely create and destroy empty lock groups:

$$\mathbf{newgroup}_{\mathrm{G}} : \mathbf{1} \to \mathbf{Lock}_{\mathrm{G}}\langle\rangle$$
$$\mathbf{dropgroup}_{\mathrm{G}} : \mathbf{Lock}_{\mathrm{G}}\langle\rangle \to \mathbf{1}$$

Once we have a lock group, we are able to create a new lock in the lock group, choosing at which position i to place it in the list:

$$\mathbf{newlock}_{\mathrm{G}}[i] : \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \vec{g}_2 \rangle \to \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \tau^1_1, \vec{g}_2 \rangle \qquad \text{where } i = |\vec{g}_1|$$

Similarly, we are able to drop a lock from the group, provided it is a client reference:

$$\mathbf{droplock}_{\mathrm{G}}[i] : \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \tau^0_0, \vec{g}_2 \rangle \to \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \vec{g}_2 \rangle \qquad \text{where } i = |\vec{g}_1|$$

The more interesting operation is acquire, which acquires one of the locks in the group:

$$\mathbf{acquire}_{\mathrm{G}}[i] : \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \tau^a_0, \vec{g}_2 \rangle \to \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \tau^a_1, \vec{g}_2 \rangle \times \tau \qquad \text{where } i = |\vec{g}_1| \text{ and } \mathbf{closed}\ \vec{g}_2$$

Acquire requires that we obey the lock order, that is, we cannot **acquire** a lock in the group if there is an opened lock to its right in the type-level list. This condition is necessary to prevent **acquire-acquire** deadlocks. The rule for release is as folows:

$$\mathbf{release}_{\mathrm{G}}[i] : \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \tau^a_1, \vec{g}_2 \rangle \times \tau \to \mathbf{Lock}_{\mathrm{G}}\langle \vec{g}_1, \tau^a_0, \vec{g}_2 \rangle \qquad \text{where } i = |\vec{g}_1|$$

$$\frac{\Gamma \text{ unr}}{\Gamma \vdash \mathbf{newgroup_G}() : \mathbf{Lock_G}\langle\rangle} \qquad \frac{\Gamma \vdash e : \mathbf{Lock_G}\langle\rangle}{\Gamma \vdash \mathbf{dropgroup_G}(e) : \mathbf{1}}$$

$$\frac{\Gamma \vdash e : \mathbf{Lock_G}\langle \vec{g}_1, \vec{g}_2\rangle \qquad i = |\vec{g}_1|}{\Gamma \vdash \mathbf{newlock_G}[i](e) : \mathbf{Lock_G}\langle \vec{g}_1, \tau_1^1, \vec{g}_2\rangle} \qquad \frac{\Gamma \vdash e : \mathbf{Lock_G}\langle \vec{g}_1, \tau_0^0, \vec{g}_2\rangle \qquad i = |\vec{g}_1|}{\Gamma \vdash \mathbf{droplock_G}[i](e) : \mathbf{Lock_G}\langle \vec{g}_1, \vec{g}_2\rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{Lock_G}\langle \vec{g}_1, \tau_0^1, \vec{g}_2\rangle \qquad \mathbf{closed}\ \vec{g}_1, \vec{g}_2, \mathbf{owners}\ \vec{g}_2 \qquad i = |\vec{g}_1|}{\Gamma \vdash \mathbf{wait_G}[i](e) : \mathbf{Lock_G}\langle \vec{g}_1, \vec{g}_2\rangle \times \tau}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{Lock_G}\langle \vec{g}\rangle \qquad \Gamma_2 \vdash e_2 : \mathbf{Lock_G}\langle \vec{g}_1\rangle \multimap \mathbf{1} \qquad \mathbf{split}\ \vec{g}\ \mathbf{into}\ \vec{g}_1, \vec{g}_2}{\Gamma \vdash \mathbf{fork_G}(e_1, e_2) : \mathbf{Lock_G}\langle \vec{g}_2\rangle}$$

$$\frac{\Gamma \vdash e : \mathbf{Lock_G}\langle \vec{g}_1, \tau_0^a, \vec{g}_2\rangle \qquad i = |\vec{g}_1| \qquad \mathbf{closed}\ \vec{g}_2}{\Gamma \vdash \mathbf{acquire_G}[i](e) : \mathbf{Lock_G}\langle \vec{g}_1, \tau_1^a, \vec{g}_2\rangle \times \tau}$$

$$\frac{\Gamma \equiv \Gamma_1 \cdot \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \mathbf{Lock_G}\langle \vec{g}_1, \tau_1^a, \vec{g}_2\rangle \qquad \Gamma_2 \vdash e_2 : \tau \qquad i = |\vec{g}_1|}{\Gamma \vdash \mathbf{release_G}[i](e_1, e_2) : \mathbf{Lock_G}\langle \vec{g}_1, \tau_0^a, \vec{g}_2\rangle}$$

Fig. 4. $\lambda_{\text{lock++}}$'s lock group typing rules.

The condition for wait has to be even more stringent than the rule for acquire:

$$\mathbf{wait_G}[i] : \mathbf{Lock_G}\langle \vec{g}_1, \tau_0^1, \vec{g}_2\rangle \to \mathbf{Lock_G}\langle \vec{g}_1, \vec{g}_2\rangle \times \tau \qquad \text{where } \mathbf{closed}\ \vec{g}_1, \vec{g}_2, \mathbf{owners}\ \vec{g}_2 \text{ and } i = |\vec{g}_1|$$

The condition says that we can only wait if all locks are closed (to prevent **acquire-wait** deadlocks on different locks in the group), and we must obey the lock order with respect to the owners, that is, we cannot **wait** if there is a client to the right (to prevent **wait-wait** deadlocks on different locks).

The rule for fork allows us to share an entire lock group with the child thread:

$$\mathbf{fork_G} : \mathbf{Lock_G}\langle \vec{g}\rangle \times (\mathbf{Lock_G}\langle \vec{g}_1\rangle \multimap \mathbf{1}) \to \mathbf{Lock_G}\langle \vec{g}_2\rangle \qquad \text{where } \mathbf{split}\ \vec{g}\ \mathbf{into}\ \vec{g}_1, \vec{g}_2$$

The relation $\mathbf{split}\ \vec{g}\ \mathbf{into}\ \vec{g}_1, \vec{g}_2$ specifies that locks are split as in λ_{lock}, but elementwise for each lock:

$$\tau_{b_1+b_2}^{a_1+a_2} \in \vec{g} \quad \text{is split into} \quad \begin{cases} \tau_{b_1}^{a_1} \in \vec{g}_1 \\ \tau_{b_2}^{a_2} \in \vec{g}_2 \end{cases}$$

The rules for $\lambda_{\text{lock++}}$ are summarized in Figure 4. In summary, $\lambda_{\text{lock++}}$ organises locks into lock groups, which can be grown and shrunk dynamically.

6.1 Examples of using lock orders

The key improvement over λ_{lock} is that $\lambda_{\text{lock++}}$'s $\mathbf{fork_G}$ allows us to share an entire lock group, with potentially multiple locks:

$$\begin{array}{l} \mathbf{let}\ \ell\ =\ \mathbf{newgroup_G}()\ \mathbf{in} \\ \mathbf{let}\ \ell\ =\ \mathbf{newlock_G}[0](\ell)\ \mathbf{in} \\ \mathbf{let}\ \ell\ =\ \mathbf{newlock_G}[1](\ell)\ \mathbf{in} \\ \mathbf{let}\ \ell\ =\ \mathbf{fork_G}(\ell, \lambda\ell.\ \cdots)\ \mathbf{in}\ \cdots \end{array}$$

In a $\lambda_{\text{lock++}}$ version of the bank example (Section 2.3.4), this would allow us to have multiple bank threads that each do transactions over multiple locks, guaranteeing deadlock freedom because the type system ensures that the banks acquire the locks according to the lock order.

Dining philosophers. We can use lock groups for a dynamic version of Dijkstra's dining philosophers (*a.k.a.,* a unbounded process network [Giachino et al. 2014; Kobayashi and Laneve 2017]), where the number n of philosophers is chosen dynamically.

$$
\begin{array}{l}
\mathsf{dine} : \mathbf{Lock}_{\mathrm{G}}\langle \mathbf{int}_0^{a_1}, \mathbf{int}_0^{a_2} \rangle \multimap \mathbf{1} \\
\mathsf{dine}(\ell) \triangleq \\
\quad \begin{array}{l}
\mathbf{let}\ \ell, x = \mathbf{acquire}_{\mathrm{G}}[0](\ell)\ \mathbf{in} \\
\mathbf{let}\ \ell, y = \mathbf{acquire}_{\mathrm{G}}[1](\ell)\ \mathbf{in} \\
\mathbf{let}\ \ell = \mathbf{release}_{\mathrm{G}}[0](\ell, y)\ \mathbf{in} \\
\mathbf{let}\ \ell = \mathbf{release}_{\mathrm{G}}[1](\ell, x)\ \mathbf{in}\ \mathsf{dine}(\ell)
\end{array} \\
\\
\mathsf{phil} : \mathbf{int} \times \mathbf{Lock}_{\mathrm{G}}\langle \mathbf{int}_0^{a_1}, \mathbf{int}_0^{a_2} \rangle \multimap \mathbf{1} \\
\mathsf{phil}(0, \ell) \triangleq \mathsf{dine}(\ell) \\
\mathsf{phil}(n+1, \ell) \triangleq \\
\quad \begin{array}{l}
\mathbf{let}\ \ell = \mathbf{release}_{\mathrm{G}}[2](\mathbf{newlock}_{\mathrm{G}}[2](\ell), 42)\ \mathbf{in} \\
\mathbf{let}\ \ell = \mathbf{fork}_{\mathrm{G}}(\ell, \lambda\ell.\ \mathsf{dine}(\mathbf{droplock}_{\mathrm{G}}[0](\ell)))\ \mathbf{in} \\
\mathsf{phil}(n, \mathbf{droplock}_{\mathrm{G}}[1](\ell))
\end{array}
\end{array}
$$

We can start the philosophers by running the following code:

$$
\begin{array}{l}
\mathbf{let}\ \ell = \mathbf{newgroup}_{\mathrm{G}}()\ \mathbf{in} \\
\mathbf{let}\ \ell = \mathbf{release}_{\mathrm{G}}[0](\mathbf{newlock}_{\mathrm{G}}[0](\ell), 42)\ \mathbf{in} \\
\mathbf{let}\ \ell = \mathbf{release}_{\mathrm{G}}[1](\mathbf{newlock}_{\mathrm{G}}[1](\ell), 42)\ \mathbf{in} \\
\mathbf{let}\ \ell = \mathbf{fork}_{\mathrm{G}}(\ell, \lambda\ell.\ \mathsf{dine}(\ell))\ \mathbf{in}\ \mathsf{phil}(n, \ell)
\end{array}
$$

The code sets up a ring of forks (the locks) and philosophers (the dining threads) between them. It helps to think of the ring as a long line of forks, which is closed by making a philosopher dine with the first and last forks in the line. Intuitively, the phil function takes a lock group with two forks: the very first fork in the line, and the last fork in the line so far. The function then creates a new fork at the end of the line, and makes a new philosopher dine with the last two forks. We then forget about the penultimate fork, and make a recursive call. At the end of the recursion, we close the loop by making a philosopher dine with the very first fork and the last. To initialize this process, we create the first two forks in the line. We make a philosopher dine with these forks, and use phil to make the line and close the loop. Note that there will be $n + 2$ locks in the lock group overall, but the local view of any reference has at most 3 locks visible at any time.

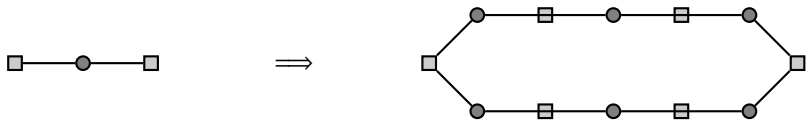
Growing the table. To illustrate the dynamic nature of lock groups and their lock orders, we can modify the above code to dynamically grow the number of philosophers at the circular table. To do so, replace the recursive call of $\mathsf{dine}(\ell)$ with $\mathsf{phil}(2, \ell)$, making dine and phil mutually recursive. Now, after a philosopher is done dining (*i.e.,* acquring and releasing their two locks), the philosopher replaces itself with 3 dining philosophers, thus growing the circle by 2.

Growing the table, fractally. To further illustrate the dynamic nature, consider replacing the recursive call $\mathsf{dine}(\ell)$ with the following code:

$$\mathsf{phil}(2, \mathbf{fork}_{\mathrm{G}}(\ell, \lambda\ell.\ \mathsf{phil}(2, \ell)))$$

After dining, the philosopher replaces itself with 6 philosophers, arranged in two parallel lines. The leftmost philosophers in the two lines both use the left-hand lock of the original philosopher, and the rightmost philosophers in the two lines both use the right-hand lock of the original philosopher.

Of course, each of these philosophers is running the same code, and after dining they will in turn replace themselves with such parallel lines, according to the following substitution:



In this picture, the squares are forks (locks) and the circles are philosophers (threads). Some forks are now accessed by more than 2 philosophers, and the number of philosophers accessing a fork can grow dynamically. Which philosopher dines and splits is non-deterministic, and thus the initial table of 42 grows non-deterministically in fractal and intricately interconnected circular patterns. Nevertheless, the meta theory of $\lambda_{\text{lock++}}$ automatically guarantees deadlock freedom by construction.

Multiple lock groups. Note that so far we have used only a single lock group. The expressiveness of λ_{lock} for multiple locks based on the sharing topology is still available for $\lambda_{\text{lock++}}$, but now for multiple lock groups. The reader may wonder how the expressivity of pure lock orders with a single lock group of $\lambda_{\text{lock++}}$ compares with the expressivity of the pure sharing topology of λ_{lock}. The two mechanisms are orthogonal, and one is not strictly more powerful than the other, because two locks in the same lock group always need to be locked in the given order, whereas two independent locks can be locked in any order.

6.2 References to lock groups

Operationally, $\lambda_{\text{lock++}}$ works precisely like λ_{lock}, except that each lock group now stores a collection of locks, each of which is identified by an id (a natural number). Each reference to a lock group may only have partial knowledge of which locks are present in the group, because new locks may have been added concurrently by other threads that hold a reference to the same lock group. However, note that the operations $\mathbf{newlock}_{\mathrm{G}}[i]$, $\mathbf{droplock}_{\mathrm{G}}[i]$, $\mathbf{acquire}_{\mathrm{G}}[i]$ and $\mathbf{release}_{\mathrm{G}}[i]$ refer to a lock by index i, which is the index of the *local view* of the lock group. Therefore, each reference to a lock group now consists not just of $\langle k \rangle$, but in fact of $\langle k \mid i_0, i_1, \ldots, i_n \rangle$, where k identifies the lock group, and $i_0, i_1, \ldots, i_n$ identifies *which* locks in the lock group this reference knows about. Thus, when we have $\langle k \mid i_0, i_1, \ldots, i_n \rangle : \mathbf{Lock}_{\mathrm{G}}\langle \vec{g} \rangle$, we have $|\vec{g}| = n$.

6.3 The invariant for lock groups

The invariant for lock groups is very similar to the one for locks. The sharing topology does not distinguish between the individual locks in a group, but treats them as an atomic whole. Thus, we may have edges between threads and lock groups, and between lock groups and lock groups, and this graph must be acyclic. The local invariant for a lock group with respect to the types $\langle k \mid i_0, i_1, \ldots, i_n \rangle : \mathbf{Lock}_{\mathrm{G}}\langle \vec{g} \rangle$ of all the references to the lock group, is also similar. Elementwise, we insist the same as for single locks: each lock must have precisely one owning reference, and the reference count of each lock must agree with the number of client references. Furthermore, whether the lock stores $\mathbf{None} \mid \mathbf{Some}(v)$ must agree with the existence of an open reference. In other words, the invariant for a lock group is the same as for single locks, but elementwise.

The key difference is that for lock groups, we insist that *the order of the lists* $I = i_0, i_1, \ldots, i_n$ of the lock references of the various references *must agree*. That is, there is some list I_{all} such that the I of every client reference $\langle k \mid I \rangle$ to the lock group is a subsequence of it.

This invariant is preserved by all the operations:

- Inserting a new lock into the group inserts it into I_{all} as well.
- Deleting a lock using $\mathbf{wait}_{\mathrm{G}}[i]$ deletes it from I_{all} as well.
- Dropping a client reference to a lock has no effect on I_{all}; we only need to ajust the subsequence witness of that reference.
- Acquiring and releasing a lock has no effect on I_{all}.
- Forking a lock group has no effect on I_{all}, since the two newly split references have the locks in the same order as the original reference.

In summary, the references to the lock group have a partial but consistent view of the lock order.

6.4 Reachability for lock groups

We wish to generalize the theorems of Section 4 to $\lambda_{\text{lock++}}$. This turns out to be very easy: only the definition of blocked (Definition 4.1) depended on the details of λ_{lock}. We adjust it so that a thread is considered blocked on a lock group if it is trying to perform one of $\lambda_{\text{lock++}}$'s lock group operations on it.

In Section 5 we have seen that the difficult case of the reachability proof is to establish reachability of the locks, or in our case now, a lock group. The reachability proof for a lock comes down to showing that it is impossible that every reference to the lock is blocked on it. Let us thus see why this also holds for lock groups. Suppose that there is some reference to the lock group. If this reference is doing anything other than acquire or wait on this lock group, then it is not blocked, and hence we're done. If it is blocked, we have the following two cases:

The case of acquire. Consider the case when this reference is doing an acquire of some lock i in the group. If this acquire can't proceed, the lock must have already been acquired, via some other reference. Consider, now, the thread holding that reference (if it is held by a lock, we are also immediately done). If that thread is not blocked on this lock group, we're done. If it *is* blocked on this lock group, it could be doing an acquire, or a wait. In fact, it is not possible that the thread is doing a wait operation, because the typing rule of wait says that *all the locks must be closed*, and if it has acquired a lock in the group, this condition is violated. Hence, the thread must be doing an acquire of some lock j in the group. It seems that we're now back to where we started. However, due to the typing rule of acquire, the lock j must be higher in the lock order than lock i. Thus, we have made progress, in the sense that we have gone up in the lock order. Hence, if we keep repeating this reasoning, we go up and up in the lock order, until we're at the end, and then the thread can't be doing any acquire (formally, we phrase this using induction). In summary, if some thread is doing an acquire, then there must be some reference into the lock group that is not blocked on the lock group, and we're done.

The case of wait. Now consider the case when the reference is doing a wait on some lock i in the group. If this wait cannot proceed, then the refcount of that lock must be nonzero, so there must also be some client reference to i. Consider, now, the thread holding that reference (if it is held by a lock, we are also immediately done). If that thread is not blocked on this lock group, we're done. If it *is* blocked on this lock group, it could be doing an acquire, or a wait. If it's doing an acquire, we're done, by the preceding paragraph. Hence, suppose that the thread is doing a wait on some lock j in the group. It seems that we're now back to where we started. However, due to the typing rule of wait, the lock j must be higher in the lock order than lock i. Hence, by repeating this

reasoning, we go up in the lock order, and eventually we're done. In summary, if some thread is doing a wait, then there must be some reference into the lock group that is not blocked on the lock group, and we're done.

7 MECHANIZED PROOFS

All of our theorems (Theorem 4.2, Theorem 4.3, Theorem 4.4, Theorem 5.1, and Theorem 5.2) have been mechanized in the Coq proof assistant [Jacobs and Balzer 2022], for both λ_{lock} and $\lambda_{\text{lock++}}$. The mechanization is structured as follows:

- The λ_{lock}/$\lambda_{\text{lock++}}$ language definition: expressions, static type system (with unrestricted and recursive types), and operational semantics.
- The configuration invariant, stating that the configuration remains well typed, that the sharing topology is acyclic, and that the lock invariants hold for every lock / lock group.
- Proof that the invariant is preserved by the operational semantics (Theorem 5.1).
- Proof that configurations satifsying the invariant are fully reachable (Theorem 5.2).
- Proofs that full-reachability is equivalent to deadlock/leak freedom, and that they imply type safety and global progress (Theorem 4.2, Theorem 4.3).

In order to handle recursive types, we use the coinductive method of Gay et al. [2020].

The mechanization uses a graph library to reason about the graph underlying the sharing topology [Jacobs et al. 2022a] and depends on Iris, mainly for the Iris proof mode [Jung et al. 2018b, 2015; Krebbers et al. 2017], as well as on the stdpp extended standard library for Coq [Coq-std++ Team 2021].

8 RELATED WORK

Related work on deadlock freedom spans both shared memory and message-passing concurrency as well as type systems and program logics. Related work on memory leak freedom seems to be confined to the purely linear setting. While memory safety has been studied both in research [Grossman et al. 2002; Tofte and Talpin 1997] and in practice, with Rust as the most prominent example [Jung et al. 2018a], memory safety does not entail memory leak freedom.

Session types. Conceptually, our work is most closely related to the family of binary session types [Caires et al. 2016; Fowler et al. 2021, 2019; Jacobs et al. 2022a; Kokke et al. 2019; Lindley and Morris 2015, 2016, 2017; Toninho 2015; Toninho et al. 2013] that build on the Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012]. Like these systems, our type system uses linearity to restrict the propagation of references to rule out circular waiting dependencies among a program's run-time objects. Our **fork** construct, moreover, resembles process spawning (*a.k.a.,* cut) in that it connects a parent and a child thread with exactly one lock (group). However, our **fork** construct differs in that it allows a parent thread to *share* the same lock (group) among repeatedly forked off children (*e.g.,* example in Section 2.3.3), permitting aliases to a lock (group) to exist and threads with such aliases to affect each other. We remark that the linear exponential, supported by some linear session types, has a copying and not a sharing semantics.

Traditional session types, both binary [Honda 1993; Honda et al. 1998] and multiparty [Honda et al. 2008], suffer from deadlock. Carbone and Debois [2010] were the first to explore the benefits of acyclicity of the underlying communication topology for deadlock freedom. These ideas, combined with insights gained from the Curry-Howard correspondence between linear logic and the session-typed π-calculus [Caires and Pfenning 2010; Wadler 2012], gave rise to a series of work to establish deadlock freedom for multiparty session types [Carbone et al. 2016, 2015, 2017; Castro-Perez et al.

2021; Jacobs et al. 2022b]. While our notion of sharing topology draws inspiration from these works, our type system offers unrestricted sharing through locks.

Recently, Qian et al. [2021] and Rocha and Caires [2021] have inhaled the linear exponential a slightly different semantics. In particular, Qian et al. [2021] observe that the established interpretation of the linear exponential [Wadler 2012] fails to faithfully encode client-server interactions, where clients are served in a non-deterministic fashion. The authors complement the linear exponential with a coexponential to fill this gap. Like acquires in our language, Qian et al. [2021]'s coexponential is the source of non-determinism. However, the coexponential still has a copying semantics, ruling out various sharing scenarios, such as dining philosophers (see Section 6.1).

Closer to our base calculus λ_{lock} is Rocha and Caires [2021]'s PaT language with reference cells. PaT's reference cells have constructs for reading a cell's contents, updating it, and locking it, while remaining deadlock-free. To account for the non-determinism resulting from an update, the authors introduce non-deterministic sums from differential linear logic [Ehrhard 2018; Ehrhard and Regnier 2006]. A similarity between PaT and λ_{lock} is the reliance on acyclicity for deadlock freedom not just for sessions, but also for reference cells. PaT ensures acyclicity with co-contraction rules for its **share** construct, which serves a similar purpose as λ_{lock}'s **fork**, with the difference that PaT is situated in a π-calculus, rather than a λ-calculus like λ_{lock}. Like **fork**, the typing rules of **share** distribute the lock's open/closed state over parallel processes, ensuring that only a single process is interacting with an opened lock. In contrast to PaT's reference cells, which can only store unrestricted values (*i.e.,* values that can be freely copied and discarded, such as natural numbers), λ_{lock}'s locks can store arbitrary linear values, including values representing non-affine obligations. Locking and unlocking transfers full ownership over the contents, including obligations, such as the obligation to close a lock or send a message on a channel. In terms of our invariants (Section 2, principles 1-5, and Section 5), these obligations emerge as edges between two locks as well as the need to introduce the owner/client distinction in addition to the open/closed distinction. Moreover, our extended language $\lambda_{\text{lock}_{++}}$ supports cyclic sharing topologies, which are beyond the reach of reference cells.

Among the extensions of linear logic session types with non-determinism and notions of sharing, manifest sharing [Balzer and Pfenning 2017; Balzer et al. 2018, 2019] is the work closest to ours. Manifest sharing introduces an adjoint formulation of linear and shared session types, with the latter resembling our locks, which can be freely shared and must be communicated with by entering a critical section. Mutual exclusion is enforced by adjoint modalities, with an acquire and release semantics. While the original system [Balzer and Pfenning 2017; Balzer et al. 2018] can suffer from deadlocks, Balzer et al. [2019] augment manifest sharing with partial orders to rule out deadlocks. In contrast to our locks, shared processes in [Balzer et al. 2019] cannot store any linear resources. Moreover, while Balzer et al. [2019]'s system supports order-polymorphic processes, ensuring compositionality, local orders must comply with a global order at run-time, whereas lock group orders in $\lambda_{\text{lock}_{++}}$ are independent of each other. Lastly, Balzer et al. [2019]'s system does not support unbounded process networks (see Section 6.1), whereas $\lambda_{\text{lock}_{++}}$ does.

Usages and obligations. The addition of channel usage information to types in a concurrent, message-passing setting was pioneered by Igarashi and Kobayashi [1997]; Kobayashi [1997], who applied the idea to deadlock prevention in the π-calculus as well as race freedom [Igarashi and Kobayashi 2001, 2004]. Typically, types are augmented with the relative ordering of channel actions, with the type system ensuring that the transitive closure of such orderings forms a strict partial order. Building on this, Kobayashi [2002a] proposed type systems that ensure a stronger property, dubbed lock freedom, and variants that are amenable to type inference [Kobayashi 2005; Kobayashi et al. 2000]. Kobayashi [2006] extended this to account for recursive processes and type inference.

The most advanced system [Giachino et al. 2014; Kobayashi and Laneve 2017] in this series supports unbounded process networks, allowing dynamic creation of circular topologies.

Padovani [2014] contributes a simplified account of Kobayashi-style orders, albeit at the cost of expressivity, which, assuming linear channel usage, gets by with a single priority rather than usage information. Padovani [2014]'s system supports priority polymorphism to support cyclic interleavings of recursive processes. Padovani's system also served as a source of inspiration for the development of a functional language with session types by Dardha and Gay [2018]; Kokke and Dardha [2021]. The authors' system focuses on the integration with a functional language, and currently lacks support of recursive circular behavior. Kobayashi-style orders have also been adopted in the multiparty session type setting [Bettini et al. 2008; Coppo et al. 2013, 2016] to establish global progress in the presence of multiparty session interleavings.

Like Giachino et al. [2014]; Kobayashi and Laneve [2017]'s system, our extended system $\lambda_{\text{lock}_{++}}$ supports unbounded process networks (see Section 6.1). However, the two systems differ conceptually: whereas our approach is primarily guided by topology, Kobayashi-style orders are guided by orders. As a result, the systems differ in technical details and user experience. For example, our core language λ_{lock} does not require any additional annotations, deadlock freedom simply follows from thread-local linearity. On the other hand, $\lambda_{\text{lock}_{++}}$ does require lock orders. These orders, however, are purely local to a lock group, and there is no need for local orders to comply with each other or a global lock order, or any other condition across groups, when acquiring locks from distinct groups. To the best of our knowledge, this feature is novel and increases compositionality.

Program logics. Our work is tangentially related to works using Hoare logics with lock orders [Hamin and Jacobs 2018; Leino et al. 2010] for deadlock freedom and work using concurrent separation logic [da Rocha Pinto et al. 2014; D'Osualdo et al. 2021; Farka et al. 2021; Jung et al. 2018b, 2015; Nanevski et al. 2019] for program verification. Our focus is on type-based, and thus automated verification. A fully fledged separation logic that is capable of proving deadlock freedom using sharing topologies is still missing. This is something we hope to explore in the future.

9 LIMITATIONS AND FUTURE WORK

No decidable type system is without its limitations, and $\lambda_{\text{lock}_{++}}$ is no exception. To our understanding, the main limitations of λ_{lock} and $\lambda_{\text{lock}_{++}}$ are as follows:

Lock group references have static size. While locks can be added to a lock group dynamically, controlled by a run-time variable n (e.g., in dynamic dining philosophers), the number of locks that can be accessed *via a single lock group reference* at any given point in the program is statically determined by the length of the type-level list (*e.g.,* in dining philosophers, each philosopher accesses two locks). This type dependency curtails expressivity when a lock group reference is used in a loop, requiring the type to be invariant across iterations and thus fixing simultaneous access to a statically predetermined number of locks in a lock group, rather than adjusting that number dynamically.

DAG-shaped mutable data structures. The simple locks of λ_{lock} can be used as mutable reference cells. The operational semantics employs reference counting memory management. Reference counting guarantees memory leak freedom as long as the data has the shape of a directed acyclic graph (DAG), and is often used that way. In a DAG, as opposed to a tree, a node may have multiple parents. In λ_{lock}, a node can have multiple parents as well, but these parents are always *disjoint*. Thus, in terms of data shapes that can be expressed, λ_{lock} supports a strict superset of tree-shapes, but a strict subset of DAG-shapes. It would be nice to relax this restriction and support general DAGs, but we do not know how to do so without introducing the possibility of deadlock. The issue

is that we could potentially obtain a duplicate reference to the same lock via different paths in the DAG, which can be used to create deadlocks.

Rust's **unsafe**. Rust has the **unsafe** mechanism for code that violates the rules of the borrow checker, to be used if the programmer promises that the code is safe and upholds Rust's invariants [Jung et al. 2018a]. One could analogously imagine an **unsafe** construct for λ_{lock} that allows the programmer to violate the linearity restriction and freely duplicate a lock, if the programmer promises that the code is safe and upholds λ_{lock}'s invariants. The open problem would be characterizing the invariants that the programmer would be responsible for upholding in their unsafe code, such that deadlock and leak freedom is guaranteed even when their unsafe code is mixed with other code. A mechanized meta theory will require extending tools like Iris with support for deadlock and leak freedom based on sharing topologies.

10 CONCLUSION

We have presented λ_{lock}, a language with locks where deadlock and memory leak freedom is guaranteed by type checking. Deadlock and leak freedom are ensured by restricting the sharing topology between locks and threads. This enables the λ_{lock} type system to be free of additional checks, such as lock orders.

The locks in λ_{lock} are *higher-order*, meaning that we can store arbitary linear values in locks, and locks themselves are completely first class entities. In particular, we can store locks in locks. This is a crucial ingredient that allows us to implement session-typed channels in terms of locks.

We have also presented $\lambda_{\text{lock++}}$, which extends λ_{lock} with *lock groups*, and allows sharing multiple locks with the child thread when we fork. This is kept deadlock free by requiring the **acquire** and **wait** operations to happen in accordance with the lock order local to the group. Crucially, there is no global order, and different lock groups can be interacted with completely independently; locks from different lock groups can be acquired simultaneously without restrictions.

We hope that this is a step toward the end goal of having an expressive concurrent language where deadlock and leak freedom follow from the type system. As a next step toward this, we would like to distill more general principles that allow us to design concurrent languages based on the sharing topology.

ACKNOWLEDGMENTS

We thank the anonymous referees for their valuable feedback and suggestions, and we thank Robbert Krebbers, Ike Mulder, Anton Golov, Jorge Pérez, Bas van den Heuvel, Dan Frumin, Luís Caires, and Pedro Rocha for helpful discussions. Stephanie Balzer was supported in part by AFOSR under grant FA9550-21-1-0385 (Tristan Nguyen, program manager) and by the National Science Foundation under award number CCF-2211996. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR or NSF.

REFERENCES

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. *PACMPL* 1, ICFP (2017), 37:1–37:29. https://doi.org/10.1145/3110281

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *CONCUR (LIPIcs, Vol. 118)*. 30:1–30:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP (LNCS, Vol. 11423)*. 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR (LNCS, Vol. 5201)*. 418–433. https://doi.org/10.1007/978-3-540-85361-9_33

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR (LNCS, Vol. 6269)*. 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear Logic Propositions as Session Types. *Math. Struct. Comput. Sci.* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

Marco Carbone and Søren Debois. 2010. A Graphical Approach to Progress for Structured Communication in Web Services. In *ICE (EPTCS, Vol. 38)*. 13–27. https://doi.org/10.4204/EPTCS.38.4

Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR (LIPIcs, Vol. 59)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2016.33

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2015. Multiparty Session Types as Coherence Proofs. In *CONCUR (LIPIcs, Vol. 42)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 412–426. https://doi.org/10.4230/LIPIcs.CONCUR.2015.412

Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Informatica* 54, 3 (2017), 243–269. https://doi.org/10.1007/s00236-016-0285-y

David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *PLDI*. 237–251. https://doi.org/10.1145/3453483.3454041

David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. ACM, 48–64. https://doi.org/10.1145/286936.286947

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *COORDINATION*. https://doi.org/10.1007/978-3-642-38493-6_4

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

The Coq-std++ Team. 2021. An extended "standard library" for Coq. Available online at https://gitlab.mpi-sws.org/iris/stdpp.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *ECOOP (Lecture Notes in Computer Science, Vol. 8586)*. Springer, 207–231. https://doi.org/10.1007/978-3-662-44202-9_9

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS (LNCS, Vol. 10803)*. 91–109. https://doi.org/10.1007/978-3-319-89366-2_5

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *PPDP'12*. https://doi.org/10.1145/2370776.2370794

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2017. Session types revisited. *Inf. Comput.* 256 (2017), 253–286. https://doi.org/10.1016/j.ic.2017.06.002

Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. 2021. TaDA Live: Compositional Reasoning for Termination of Fine-grained Concurrent Programs. *TOPLAS* 43, 4 (2021), 16:1–16:134. https://doi.org/10.1145/3477082

Thomas Ehrhard. 2018. An Introduction to Differential Linear Logic: Proof-Nets, Models and Antiderivatives. *Math. Struct. Comput. Sci.* 28, 7 (2018), 995–1060. https://doi.org/10.1017/S0960129516000372

Thomas Ehrhard and Laurent Regnier. 2006. Differential Interaction Nets. *Theor. Comput. Sci.* 364, 2 (2006), 166–195. https://doi.org/10.1016/j.tcs.2006.08.003

Frantisek Farka, Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2021. On Algebraic Abstractions for Concurrent Separation Logics. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434286

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. In *CONCUR (LIPIcs, Vol. 203)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:18. https://doi.org/10.4230/LIPIcs.CONCUR.2021.36

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types Without Tiers. *PACMPL* 3, POPL (2019), 28:1–28:29. https://doi.org/10.1145/3290341

Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *PLACES (EPTCS, Vol. 314)*. 23–33. https://doi.org/10.4204/EPTCS.314.3

Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. 2014. Deadlock Analysis of Unbounded Process Networks. In *CONCUR (Lecture Notes in Computer Science, Vol. 8704)*. Springer, 63–77. https://doi.org/10.1007/978-3-662-44584-6_6

Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *PLDI*. ACM, 282–293. https://doi.org/10.1145/512529.512563

Jafar Hamin and Bart Jacobs. 2018. Deadlock-Free Monitors. In *ESOP (LNCS, Vol. 10801)*. 415–441. https://doi.org/10.1007/978-3-319-89884-1_15

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR (LNCS, Vol. 715)*. 509–523. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP (LNCS, Vol. 1381)*. 122–138. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. 273–284. https://doi.org/10.1145/1328438.1328472

Atsushi Igarashi and Naoki Kobayashi. 1997. Type-Based Analysis of Communication for Concurrent Programming Languages. In *SAS (LNCS, Vol. 1302)*. 187–201. https://doi.org/10.1007/BFb0032742

Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *POPL*. 128–141. https://doi.org/10.1145/360204.360215

Atsushi Igarashi and Naoki Kobayashi. 2004. A Generic Type System for the Pi-calculus. *Theoretical Computer Science* 311, 1-3 (2004), 121–163. https://doi.org/10.1016/S0304-3975(03)00325-6

Jules Jacobs and Stephanie Balzer. 2022. Higher-Order Leak and Deadlock Free Locks (Coq mechanization). https://doi.org/10.5281/zenodo.7150549 The most recent version is at https://github.com/julesjacobs/cgraphs..

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022a. Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–33. https://doi.org/10.1145/3498662

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022b. Multiparty GV: Functional Multiparty Session Types With Certified Deadlock Freedom. *Proc. ACM Program. Lang.* ICFP (2022). To appear.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris From the Ground Up: A Modular Foundation for Higher-Order Concurrent Separation Logic. *JFP* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. 637–650. https://doi.org/10.1145/2676726.2676980

Naoki Kobayashi. 1997. A Partially Deadlock-Free Typed Process Calculus. In *LICS*. 128–139. https://doi.org/10.1109/LICS.1997.614941

Naoki Kobayashi. 2002a. A Type System for Lock-Free Processes. *I&C* 177, 2 (2002), 122–159. https://doi.org/10.1006/inco.2002.3171

Naoki Kobayashi. 2002b. Type Systems for Concurrent Programs *(Lecture Notes in Computer Science, Vol. 2757)*. 439–453. https://doi.org/10.1007/978-3-540-40007-3_26

Naoki Kobayashi. 2005. Type-Based Information Flow Analysis for the Pi-Calculus. *Acta Informatica* 42, 4-5 (2005), 291–347. https://doi.org/10.1007/s00236-005-0179-x

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR (LNCS, Vol. 4137)*. 233–247. https://doi.org/10.1007/11817949_16

Naoki Kobayashi and Cosimo Laneve. 2017. Deadlock Analysis of Unbounded Process Networks. *Inf. Comput.* 252 (2017), 48–70. https://doi.org/10.1016/j.ic.2016.03.004

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *TOPLAS* 21, 5 (1999), 914–947. https://doi.org/10.1145/330249.330251

Naoki Kobayashi, Shin Saito, and Eijiro Sumii. 2000. An Implicitly-Typed Deadlock-Free Process Calculus. In *CONCUR (LNCS, Vol. 1877)*. 489–503. https://doi.org/10.1007/3-540-44618-4_35

Wen Kokke and Ornela Dardha. 2021. Prioritise the Best Variation. In *FORTE (LNCS, Vol. 12719)*. Springer, 100–119. https://doi.org/10.1007/978-3-030-78089-0_6

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better Late Than Never: a Fully-Abstract Semantics for Classical Processes. *PACMPL* 3, POPL (2019), 24:1–24:29. https://doi.org/10.1145/3290337

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. 205–217. https://doi.org/10.1145/3009837.3009855

K. Rustan M. Leino, Peter Müller, and Jan Smans. 2010. Deadlock-Free Channels and Locks. In *ESOP (Lecture Notes in Computer Science, Vol. 6012)*. Springer, 407–426. https://doi.org/10.1007/978-3-642-11957-6_22

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP (LNCS, Vol. 9032)*. 560–584. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016. Talking Bananas: Structural Recursion For Session Types. In *ICFP*. 434–447. https://doi.org/10.1145/2951913.2951921

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*.

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust language. In *HILT*. ACM, 103–104. https://doi.org/10.1145/2663171.2663188

Peter Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. Lecture Notes in Computer Science, Vol. 2262. Springer. https://doi.org/10.1007/3-540-45651-1

Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. 2019. Specifying Concurrent Programs in Separation Logic: Morphisms and Simulations. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 161:1–161:30. https://doi.org/10.1145/3360587

Luca Padovani. 2014. Deadlock and lock freedom in the linear π-calculus. In *LICS*. ACM, 72:1–72:10. https://doi.org/10.1145/2603088.2603116

Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-Server Sessions in Linear Logic. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–31. https://doi.org/10.1145/3473567

Pedro Rocha and Luís Caires. 2021. Propositions-as-Types and Shared State. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473584

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Inf. Comput.* 132, 2 (1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation.* Ph. D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP (LNCS, Vol. 7792)*. 350–369. https://doi.org/10.1007/978-3-642-37036-6_20

Philip Wadler. 2012. Propositions as Sessions. In *ICFP*. 273–286. https://doi.org/10.1145/2364527.2364568

Received 2022-07-07; accepted 2022-11-07