Higher-Order Leak and Deadlock Free Locks

Jules Jacobs
Radboud University

Stephanie Balzer
Carnegie Mellon University

Memory management with substructural types

```
fn min(x: u32, y: u32) → u32 {
  let mut v = Vec::new();
  v.push(x);
  v.push(y);
  v.sort();
  return v[0];
  // v is deallocated
}
```

- Each heap allocation has a single owning reference
- Deallocated when owning reference disappears
- Prevents memory leaks...?

Memory leaks in Rust

`Arc<Mutex<T>>`

- Shareable mutable reference to `T`
  - Guarded by a lock
  - Reference-counted
- *"Higher-order":* can store mutexes in mutexes

```
enum List { Nil, Cons(u32, Arc<Mutex<List>>) }
```

Memory leaks in Rust

`Arc<Mutex<T>>`

- Shareable mutable reference to `T`
  - Guarded by a lock
  - Reference-counted
- *"Higher-order":* can store mutexes in mutexes

```
enum List { Nil, Cons(u32, Arc<Mutex<List>>) }
```

Memory leaks!

```
let x = Arc::new(Mutex::new(List::Nil)); // create list

*x.lock() = List::Cons(1, x.clone()); // create cycle

// refcount=2
drop(x);
// refcount=1 → list is leaked
```

Deadlocks in Rust

```
fn swap(x : &Mutex<u32>, y : &Mutex<u32>){
  let mut gx = x.lock(); // acquire locks
  let mut gy = y.lock();

  let tmp = *gx; // swap contents
  *gx = *gy;
  *gy = tmp;

  drop(gx); // release locks
  drop(gy);
}
```

Deadlocks!

```
spawn(||{ swap(x,y) });
spawn(||{ swap(y,x) });
```

The paper in a nutshell

Can we get leak and deadlock freedom by type checking?

The paper in a nutshell

Can we get leak and deadlock freedom by type checking?

Language $\lambda_{\mathbf{lock}}$ with a *linearly typed lock API*

- Any lock in scope can be safely acquired
- "Higher-order locks": can store locks in locks (recursively)
- *Sharing topology* remains acyclic by typing
- No leaks/deadlocks (✓ **in Coq**)

The paper in a nutshell

Can we get leak and deadlock freedom by type checking?

Language $\lambda_{\mathbf{lock}}$ **with a** *linearly typed lock API*

- Any lock in scope can be safely acquired
- "Higher-order locks": can store locks in locks (recursively)
- *Sharing topology* remains acyclic by typing
- No leaks/deadlocks (✓ **in Coq**)

Extension $\lambda_{\mathbf{lock{+}{+}}}$ **with** *cyclic sharing topology*

- *Lock groups* with *local lock orders*
- Locks across groups can still be acquired in arbitrary order
- No leaks/deadlocks (✓ **in Coq**)

λ_{lock}'s lock type

$$\mathbf{Lock}\langle \tau^{a}_{b} \rangle \qquad \begin{array}{l} a \in \{0,1\} \\ b \in \{0,1\} \end{array}$$

A shareable reference to τ, similar to `Arc<Mutex<t>>` in Rust

λ_{lock}'s lock type

$$\langle \tau^{a}_{b} \rangle \qquad \begin{array}{l} a \in \{0,1\} \\ b \in \{0,1\} \end{array}$$

A shareable reference to τ, similar to `Arc<Mutex<t>>` in Rust

λ_{lock}'s lock type

$$\langle \tau^{a}_{b} \rangle \qquad a \in \{0,1\} \quad b \in \{0,1\}$$

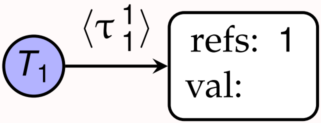
A shareable reference to τ, similar to `Arc<Mutex<t>>` in Rust

- **But** $\ell : \langle \tau^{a}_{b} \rangle$ is *linear*
- $a = 1$: this reference has to deallocate the lock
- $b = 1$: this reference has to release the lock

λ_{lock}'s lock API

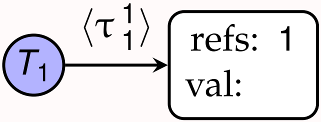
$\textbf{new} : \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$ (initially empty)

λ_{lock}'s lock API



$\textbf{new} : \ \mathbf{1} \multimap \langle \tau \,{}^1_1 \rangle$ (initially empty)

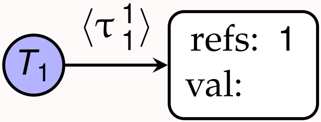
λ_{lock}'s lock API



$$\textbf{new}: \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

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

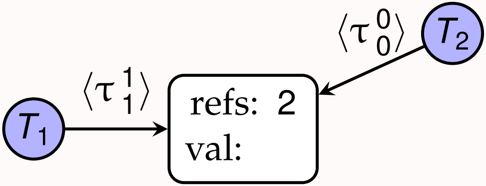
λ_{lock}'s lock API



$$\textbf{new} : \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

$$\textbf{fork} : \quad \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}}$$

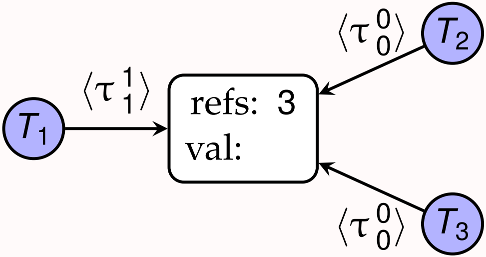
λ_{lock}'s lock API



$$\textbf{new}: \quad \mathbf{1} \multimap \langle\tau^{1}_{1}\rangle \qquad \text{(initially empty)}$$

$$\textbf{fork}: \quad \langle\tau^{a_1+a_2}_{b_1+b_2}\rangle \times (\underbrace{\langle\tau^{a_2}_{b_2}\rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle\tau^{a_1}_{b_1}\rangle}_{\text{to parent thread}}$$

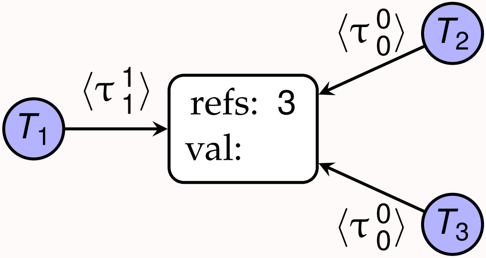
λ_{lock}'s lock API



$$\textbf{new}: \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

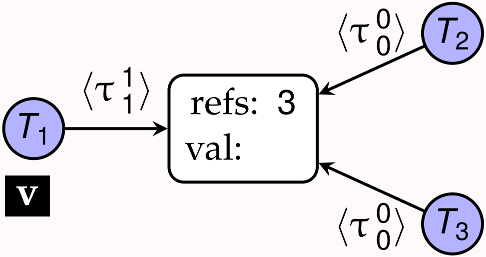
$$\textbf{fork}: \quad \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}}$$

λ_{lock}'s lock API



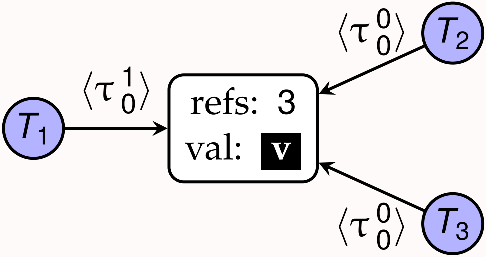
$$\begin{aligned}
\textbf{new}: &\quad \mathbf{1} \multimap \langle\tau^{1}_{1}\rangle \qquad \text{(initially empty)}\\
\textbf{fork}: &\quad \langle\tau^{a_1+a_2}_{b_1+b_2}\rangle \times (\underbrace{\langle\tau^{a_2}_{b_2}\rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle\tau^{a_1}_{b_1}\rangle}_{\text{to parent thread}}\\
\textbf{release}: &\quad \langle\tau^{a}_{1}\rangle \times \tau \multimap \langle\tau^{a}_{0}\rangle\\
\textbf{acquire}: &\quad \langle\tau^{a}_{0}\rangle \multimap \langle\tau^{a}_{1}\rangle \times \tau
\end{aligned}$$

λ_{lock}'s lock API



$$
\begin{aligned}
\textbf{new}: &\quad \mathbf{1} \multimap \langle\tau{}^{1}_{1}\rangle \qquad \text{(initially empty)} \\
\textbf{fork}: &\quad \langle\tau{}^{a_1+a_2}_{b_1+b_2}\rangle \times (\underbrace{\langle\tau{}^{a_2}_{b_2}\rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle\tau{}^{a_1}_{b_1}\rangle}_{\text{to parent thread}} \\
\textbf{release}: &\quad \langle\tau{}^{a}_{1}\rangle \times \tau \multimap \langle\tau{}^{a}_{0}\rangle \\
\textbf{acquire}: &\quad \langle\tau{}^{a}_{0}\rangle \multimap \langle\tau{}^{a}_{1}\rangle \times \tau
\end{aligned}
$$

λ_{lock}'s lock API



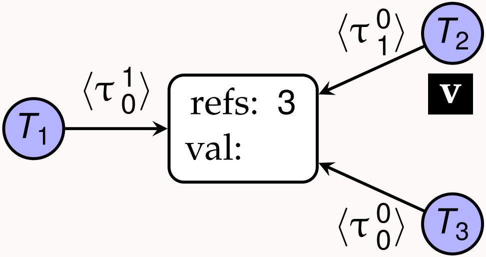
$$\textbf{new}: \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

$$\textbf{fork}: \quad \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}}$$

$$\textbf{release}: \quad \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

$$\textbf{acquire}: \quad \langle \tau^{a}_{0} \rangle \multimap \langle \tau^{a}_{1} \rangle \times \tau$$

λ_{lock}'s lock API



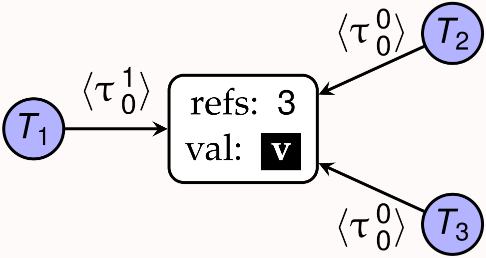
$$\textbf{new}: \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

$$\textbf{fork}: \quad \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}}$$

$$\textbf{release}: \quad \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

$$\textbf{acquire}: \quad \langle \tau^{a}_{0} \rangle \multimap \langle \tau^{a}_{1} \rangle \times \tau$$

λ_{lock}'s lock API



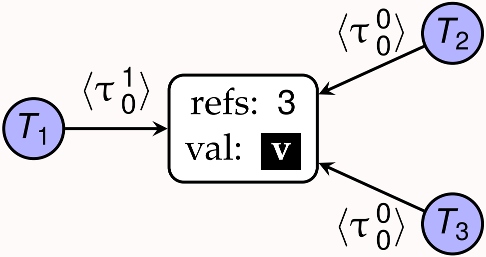
$$\textbf{new}: \quad \mathbf{1} \multimap \langle\tau^{1}_{1}\rangle \qquad \text{(initially empty)}$$

$$\textbf{fork}: \quad \langle\tau^{a_1+a_2}_{b_1+b_2}\rangle \times (\underbrace{\langle\tau^{a_2}_{b_2}\rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle\tau^{a_1}_{b_1}\rangle}_{\text{to parent thread}}$$

$$\textbf{release}: \quad \langle\tau^{a}_{1}\rangle \times \tau \multimap \langle\tau^{a}_{0}\rangle$$

$$\textbf{acquire}: \quad \langle\tau^{a}_{0}\rangle \multimap \langle\tau^{a}_{1}\rangle \times \tau$$

λ_{lock}'s lock API



$$\mathbf{new}: \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

$$\mathbf{fork}: \quad \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}}$$

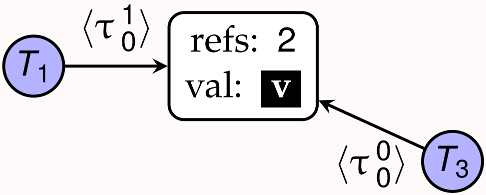
$$\mathbf{release}: \quad \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

$$\mathbf{acquire}: \quad \langle \tau^{a}_{0} \rangle \multimap \langle \tau^{a}_{1} \rangle \times \tau$$

$$\mathbf{drop}: \quad \langle \tau^{0}_{0} \rangle \multimap \mathbf{1} \qquad \text{(decrements refcount)}$$

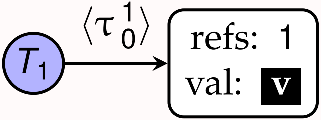
$$\mathbf{wait}: \quad \langle \tau^{1}_{0} \rangle \multimap \tau \qquad \text{(blocks until refcount} = 1)$$

λ_{lock}'s lock API



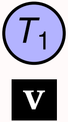
$$\begin{aligned}
\textbf{new} &: \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle && \text{(initially empty)} \\
\textbf{fork} &: \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}} \\
\textbf{release} &: \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle \\
\textbf{acquire} &: \langle \tau^{a}_{0} \rangle \multimap \langle \tau^{a}_{1} \rangle \times \tau \\
\\
\textbf{drop} &: \langle \tau^{0}_{0} \rangle \multimap \mathbf{1} && \text{(decrements refcount)} \\
\textbf{wait} &: \langle \tau^{1}_{0} \rangle \multimap \tau && \text{(blocks until refcount = 1)}
\end{aligned}$$

λ_{lock}'s lock API



$$
\begin{aligned}
\textbf{new} &: \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle && \text{(initially empty)} \\
\textbf{fork} &: \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}} \\
\textbf{release} &: \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle \\
\textbf{acquire} &: \langle \tau^{a}_{0} \rangle \multimap \langle \tau^{a}_{1} \rangle \times \tau \\
\\
\textbf{drop} &: \langle \tau^{0}_{0} \rangle \multimap \mathbf{1} && \text{(decrements refcount)} \\
\textbf{wait} &: \langle \tau^{1}_{0} \rangle \multimap \tau && \text{(blocks until refcount = 1)}
\end{aligned}
$$

λ_{lock}'s lock API



$$\textbf{new} : \quad \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle \qquad \text{(initially empty)}$$

$$\textbf{fork} : \quad \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\underbrace{\langle \tau^{a_2}_{b_2} \rangle}_{\text{to child thread}} \multimap \mathbf{1}) \multimap \underbrace{\langle \tau^{a_1}_{b_1} \rangle}_{\text{to parent thread}}$$

$$\textbf{release} : \quad \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

$$\textbf{acquire} : \quad \langle \tau^{a}_{0} \rangle \multimap \langle \tau^{a}_{1} \rangle \times \tau$$

$$\textbf{drop} : \quad \langle \tau^{0}_{0} \rangle \multimap \mathbf{1} \qquad \text{(decrements refcount)}$$

$$\textbf{wait} : \quad \langle \tau^{1}_{0} \rangle \multimap \tau \qquad \text{(blocks until refcount = 1)}$$

Which concurrency patterns does λ_{lock} support?

Simultaneously acquire multiple locks:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{swap}\ =\ \lambda(\ell_1 : \langle\tau^{\,0}_{\,0}\rangle, \ell_2 : \langle\tau^{\,0}_{\,0}\rangle).\\
\quad \textbf{let}\ (\ell_1 : \langle\tau^{\,0}_{\,1}\rangle, x_1 : \tau)\ =\ \textbf{acquire}(\ell_1)\ \textbf{in}\\
\quad \textbf{let}\ (\ell_2 : \langle\tau^{\,0}_{\,1}\rangle, x_2 : \tau)\ =\ \textbf{acquire}(\ell_2)\ \textbf{in}\\
\quad \textbf{let}\ \ell_1 : \langle\tau^{\,0}_{\,0}\rangle\ =\ \textbf{release}(\ell_1, x_2)\ \textbf{in}\\
\quad \textbf{let}\ \ell_2 : \langle\tau^{\,0}_{\,0}\rangle\ =\ \textbf{release}(\ell_2, x_1)\ \textbf{in}\\
\quad (\ell_1, \ell_2)
\end{array}
$$

✓

deadlock free

Which concurrency patterns does λ_{lock} support?

Simultaneously acquire multiple locks:

$$
\begin{array}{l}
\textbf{let}\ \mathsf{swap}\ =\ \lambda(\ell_1 : \langle \tau^{0}_{0} \rangle, \ell_2 : \langle \tau^{0}_{0} \rangle). \\
\quad \textbf{let}\ (\ell_1 : \langle \tau^{0}_{1} \rangle, x_1 : \tau)\ =\ \textbf{acquire}(\ell_1)\ \textbf{in} \\
\quad \textbf{let}\ (\ell_2 : \langle \tau^{0}_{1} \rangle, x_2 : \tau)\ =\ \textbf{acquire}(\ell_2)\ \textbf{in} \\
\quad \textbf{let}\ \ell_1 : \langle \tau^{0}_{0} \rangle\ =\ \textbf{release}(\ell_1, x_2)\ \textbf{in} \\
\quad \textbf{let}\ \ell_2 : \langle \tau^{0}_{0} \rangle\ =\ \textbf{release}(\ell_2, x_1)\ \textbf{in} \\
\quad (\ell_1, \ell_2)
\end{array}
$$

✓

deadlock free

Futures / promises / fork-join:

$$
\begin{array}{l}
\textbf{let}\ \ell : \langle \tau^{1}_{0} \rangle\ =\ \textbf{fork}(\textbf{new}() : \langle \tau^{1}_{1} \rangle, \lambda\ell : \langle \tau^{0}_{1} \rangle. \\
\qquad \textbf{drop}(\textbf{release}(\ell, E : \tau)) \\
)\ \textbf{in}\ \cdots \textbf{wait}(\ell) \cdots
\end{array}
$$

Obligation to fulfill promise cannot be discarded

Locks in locks in λ_{lock}

Store locks in locks:

$$\textbf{release}(\ell_1 : \langle\langle\tau^{a}_{b}\rangle^{0}_{1}\rangle, \ell_2 : \langle\tau^{a}_{b}\rangle)$$

✓ **leak free**

Another thread can $\textbf{acquire}(\ell_1)$ to obtain ℓ_2

Locks in locks in λ_{lock}

Store locks in locks:

$$\textbf{release}(\ell_1 : \langle\langle \tau_{b}^{a} \rangle_{1}^{0}\rangle, \ell_2 : \langle \tau_{b}^{a} \rangle)$$

✓

leak free

Another thread can $\textbf{acquire}(\ell_1)$ to obtain ℓ_2

Recursive mutable data structures:

$$\text{tree} = \langle \mathbf{1} + \tau \times \text{tree} \times \text{tree}_{0}^{1} \rangle$$

Locks in locks in λ_{lock}

Store locks in locks:

$$\textbf{release}(\ell_1 : \langle\langle\tau^{a}_{b}\rangle^{0}_{1}\rangle, \ell_2 : \langle\tau^{a}_{b}\rangle)$$

✓ **leak free**

Another thread can $\textbf{acquire}(\ell_1)$ to obtain ℓ_2

Recursive mutable data structures:

$$\text{tree} = \langle \mathbf{1} + \tau \times \text{tree} \times \text{tree}^{1}_{0}\rangle$$

Session typed channels as a library:

$$s ::= \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \,\&\, s \mid s \oplus s \mid \text{End}_{\mathbf{!}} \mid \text{End}_{\mathbf{?}} \mid \mu x.s \mid x$$

Locks in locks in λ_{lock}

Store locks in locks:

$$\textbf{release}(\ell_1 : \langle\langle \tau^{a}_{b}\rangle^{0}_{1}\rangle, \ell_2 : \langle \tau^{a}_{b}\rangle)$$

✓

leak free

Another thread can $\textbf{acquire}(\ell_1)$ to obtain ℓ_2

Recursive mutable data structures:

$$\text{tree} = \langle \mathbf{1} + \tau \times \text{tree} \times \text{tree}^{1}_{0}\rangle$$

Session typed channels as a library:

$$s ::= \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \,\&\, s \mid s \oplus s \mid \text{End}_{\mathbf{!}} \mid \text{End}_{\mathbf{?}} \mid \mu x.s \mid x$$

Shared sessions:

$$\langle \mu x.\mathbf{!}\tau.\mathbf{?}\tau.s \oplus \text{End}_{\mathbf{!}}{}^{0}_{0}\rangle$$

From λ_{lock} to $\lambda_{\text{lock++}}$

In λ_{lock}, we can duplicate *one* lock on **fork**.
Is it sound to allow duplicating *two*?

$$\textbf{let } (\ell_1, \ell_2) \ = \ \textbf{fork}((\ell_1, \ell_2), \lambda(\ell_1, \ell_2).\ \cdots) \textbf{ in } \cdots$$

From λ_{lock} to $\lambda_{\text{lock++}}$

In λ_{lock}, we can duplicate *one* lock on **fork**.
Is it sound to allow duplicating *two*?

$$\textbf{let}\ (\ell_1, \ell_2)\ =\ \textbf{fork}((\ell_1, \ell_2), \lambda(\ell_1, \ell_2).\ \cdots)\ \textbf{in}\ \cdots$$

No, because

- Deadlock: acquiring (ℓ_1 then ℓ_2) in parallel with (ℓ_2 then ℓ_1)
- Leak: storing (ℓ_1 into ℓ_2) in parallel with (ℓ_2 into ℓ_1)

From λ_{lock} to $\lambda_{\text{lock++}}$

In λ_{lock}, we can duplicate *one* lock on **fork**.
Is it sound to allow duplicating *two*?

$$\textbf{let}\ (\ell_1, \ell_2)\ =\ \textbf{fork}((\ell_1, \ell_2), \lambda(\ell_1, \ell_2).\ \cdots)\ \textbf{in}\ \cdots$$

No, because

- Deadlock: acquiring (ℓ_1 then ℓ_2) in parallel with (ℓ_2 then ℓ_1)
- Leak: storing (ℓ_1 into ℓ_2) in parallel with (ℓ_2 into ℓ_1)

Yes, if

- We make **acquire** and **wait** follow a lock order
- We prevent storing those locks inside each other

λ_{lock++}'s lock group type

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

- We can only **acquire** and **wait** in the given order
- We can add and remove locks dynamically
- The type level list is a *local view* into a complete order.

$\lambda_{\text{lock++}}$'s lock group API

$$\textbf{newgroup} : \mathbf{1} \multimap \langle\rangle$$

$$\textbf{dropgroup} : \langle\rangle \multimap \mathbf{1}$$

$$\textbf{new}[k] : \langle A, B\rangle \multimap \langle A, \tau_1^1, B\rangle \qquad (\text{length}(A) = k)$$

$$\textbf{drop}[k] : \langle A, \tau_0^0, B\rangle \multimap \langle A, B\rangle$$

$$\textbf{release}[k] : \langle A, \tau_1^a, B\rangle \times \tau \multimap \langle A, \tau_0^a, B\rangle$$

$$\textbf{acquire}[k] : \langle A, \tau_0^a, B_0\rangle \multimap \langle A, \tau_1^a, B_0\rangle \times \tau$$

$$\textbf{wait}[k] : \langle A_0, \tau_0^1, B_0^1\rangle \multimap \langle A_0, B_0^1\rangle \times \tau$$

$$\textbf{fork} : \langle A\rangle \times (\langle B\rangle \multimap \mathbf{1}) \multimap \langle C\rangle$$

$$(\text{where } A = B \oplus C)$$

Swap within a lock group

$$\mathsf{swap} : \langle \mathsf{int}_0^0, \mathsf{int}_0^0 \rangle \multimap \langle \mathsf{int}_0^0, \mathsf{int}_0^0 \rangle$$

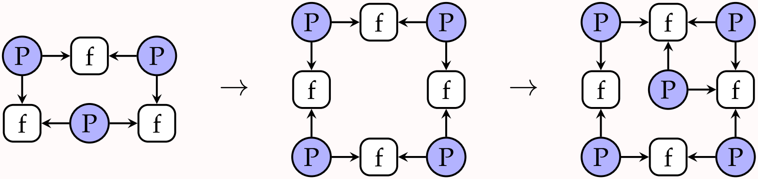
$$\begin{aligned}
&\mathsf{swap}(\ell) := \\
&\qquad \textbf{let } \ell, x \;=\; \textbf{acquire}[0](\ell) \textbf{ in} \\
&\qquad \textbf{let } \ell, y \;=\; \textbf{acquire}[1](\ell) \textbf{ in} \\
&\qquad \textbf{let } \ell \;=\; \textbf{release}[0](\ell, y) \textbf{ in} \\
&\qquad \textbf{let } \ell \;=\; \textbf{release}[1](\ell, x) \textbf{ in } \ell
\end{aligned}$$

- Type system enforces an order *within* a group
- No restrictions between two groups
  (Partial lock orders don't allow this!)

Dijkstra's dining philosophers

Lock groups allow $\lambda_{\mathbf{lock++}}$ to have cyclic connectivity

- Example: *Dijkstra's Dining Philosophers*
- Every thread (*Philosopher*) has access to 2 locks (*forks*): $\langle \mathsf{fork}^{a}_{b}, \mathsf{fork}^{a'}_{b'} \rangle$
- Can grow the dining table dynamically (fractally growing example in the paper)



Leak and deadlock freedom theorem

Small-step semantics on config $\rho = \{X_1, .., X_n\}$ of threads & locks

$X \in \rho$ waits for $Y \in \rho$ if

- X is a thread attempting an operation on lock Y, or
- Y has a reference to lock X

Leak and deadlock freedom theorem

Small-step semantics on config $\rho = \{X_1, .., X_n\}$ of threads & locks

$X \in \rho$ waits for $Y \in \rho$ if

- X is a thread attempting an operation on lock Y, or
- Y has a reference to lock X

$X \in \rho$ is *reachable* if it transitively waits for $Y \in \rho$ that can step

$S \subseteq \rho$ is a *deadlock* if no $X \in S$ can step or waits for $Y \notin S$

Leak and deadlock freedom theorem

Small-step semantics on config $\rho = \{X_1, .., X_n\}$ of threads & locks

$X \in \rho$ waits for $Y \in \rho$ if

- X is a thread attempting an operation on lock Y, or
- Y has a reference to lock X

$X \in \rho$ is *reachable* if it transitively waits for $Y \in \rho$ that can step

$S \subseteq \rho$ is a *deadlock* if no $X \in S$ can step or waits for $Y \notin S$

Theorem: all $X \in \rho$ reachable $\iff$ no deadlocks $\emptyset \subset S \subseteq \rho$

Theorem: well-typed programs are leak and deadlock free

Corollary: global progress: $\rho \neq \emptyset \to \rho$ steps

✓ **in Coq** ($\approx$ 13k loc)

Insight: leak and deadlock freedom are related

Related and future work (non-exhaustive)

Related work

- CLASS – Rocha and Caires (ICFP'21, ESOP'23)
- Client-server sessions – Qian, Kavvos, Birkedal (ICFP'21)
- Usages/obligations – Kobayashi et al. (see paper)
- Priorities – Padovani, Dharda et al. (see paper)
- Manifest sharing – Balzer et al. (ICFP'17,ESOP'19)
- Session types – (see paper)

Future work

- DAG-shaped mutable data structures / `Rc<RefCell<T>>`
- Integration with Rust features (borrowing & `unsafe`)

Rust is a *practical* memory-safe language without GC ✓

Can a *practical* language be leak and deadlock free?

I hope to have convinced you that:

- This might be possible & is worth trying
- Promising direction: single ownership $\to$ sharing topology

Rust is a *practical* memory-safe language without GC ✓

Can a *practical* language be leak and deadlock free?

I hope to have convinced you that:

- This might be possible & is worth trying
- Promising direction: single ownership $\rightarrow$ sharing topology

"The authors didn't even have to hide a bunch of more complicated rules in an appendix." – Reviewer A

(P.S. I'm looking for a postdoc position)