Higher-Order Leak and Deadlock Free Locks

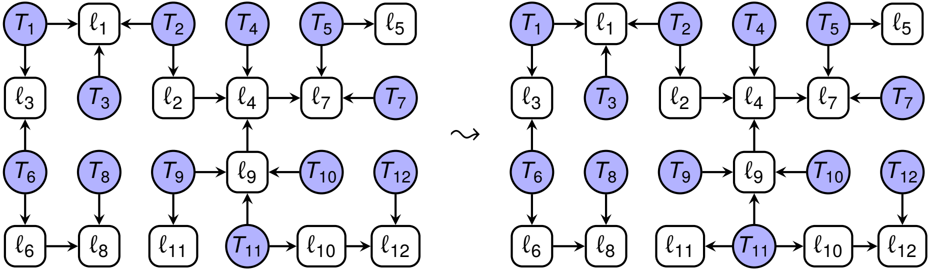
(POPL'23)

Jules Jacobs

Radboud University → Cornell 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
- 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
- Can store mutexes in mutexes

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

Memory leaks!

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

*x.lock() = 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!

```
let x = Mutex::new(1);
let y = Mutex::new(2);
fork{ swap(&x, &y); }
fork{ swap(&y, &x); }
```

Can we guarantee leak and deadlock freedom by type checking?

Yes, we can!

Language λ_{lock} with a linearly typed lock API

- No leaks/deadlocks (✓ **in Coq**)
- Any lock in scope can be safely acquired
  ```
  fn swap(x: &Mutex<u32>, y: &Mutex<u32>) ✓
  ```
- Can store locks in locks (recursively)
  ```
  enum List{Nil,Cons(u32, Arc<Mutex<List>>)} ✓
  ```
- **Key invariant:** acyclic sharing topology

Yes, we can!

Language λ_{lock} with a linearly typed lock API

- No leaks/deadlocks (✓ **in Coq**)
- Any lock in scope can be safely acquired
  ```
  fn swap(x: &Mutex<u32>, y: &Mutex<u32>)
  ```
  ✓
- Can store locks in locks (recursively)
  ```
  enum List{Nil,Cons(u32, Arc<Mutex<List>>)}
  ```
  ✓
- **Key invariant:** acyclic sharing topology

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

- No leaks/deadlocks (✓ **in Coq**)
- Cycles within **lock groups** allowed via **local lock orders** ✓
- $\lambda_{\text{lock}} \equiv$ all lock groups are singletons

λ_{lock}'s lock type

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

but *linearly typed*

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

- $a = 1$: this reference has to deallocate the lock
- $b = 1$: this reference has to release the lock

λ_{lock}'s lock type

A shareable reference to τ, similar to `Arc<Mutex<t>>` in Rust, but *linearly typed*

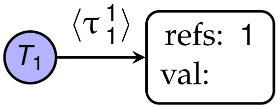
$$\langle \tau^{a}_{b} \rangle \qquad\qquad a \in \{0,1\} \quad b \in \{0,1\}$$

- $a = 1$: this reference has to deallocate the lock
- $b = 1$: this reference has to release the lock

λ_{lock}'s lock API

$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

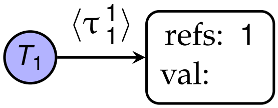
$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$



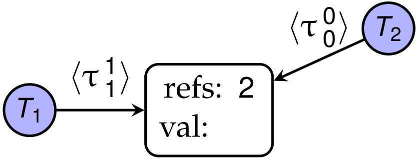
λ_{lock}'s lock API

$$\textbf{new} : \ \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

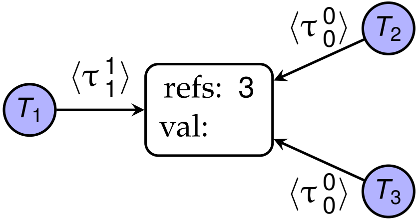


λ_{lock}'s lock API



$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

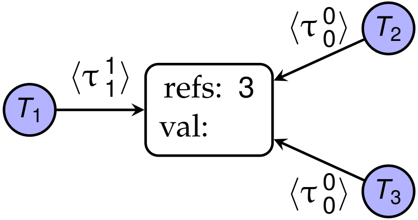
$$\textbf{fork} : \langle \tau^{a_1+a_2}_{b_1+b_2} \rangle \times (\langle \tau^{a_2}_{b_2} \rangle \multimap \mathbf{1}) \multimap \langle \tau^{a_1}_{b_1} \rangle$$



$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

λ_{lock}'s lock API



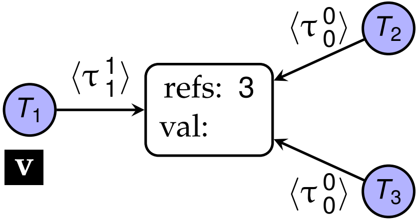
$$\textbf{new} : \ \mathbf{1} \multimap \langle\tau^{1}_{1}\rangle$$

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

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

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

λ_{lock}'s lock API



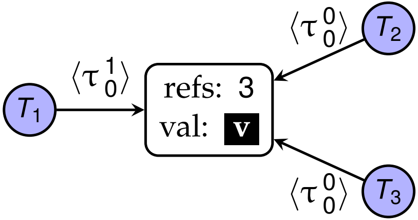
$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

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

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

λ_{lock}'s lock API

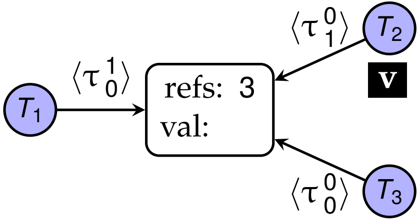


$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

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

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



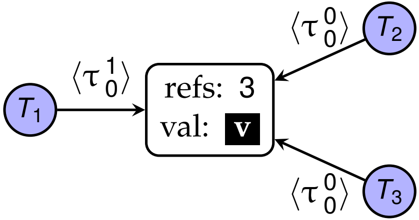
$$\textbf{new} : \mathbf{1} \multimap \langle\tau{}^{1}_{1}\rangle$$

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

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

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

λ_{lock}'s lock API



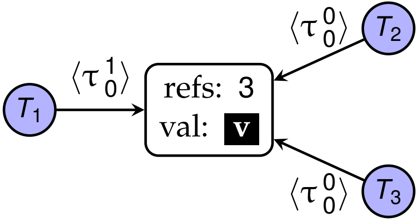
$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

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

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

λ_{lock}'s lock API



$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

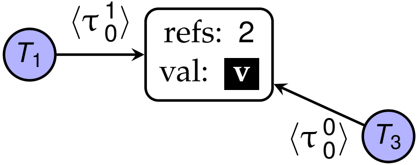
$$\textbf{release} : \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

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

$$\textbf{drop} : \langle \tau^{0}_{0} \rangle \multimap \mathbf{1}$$

$$\textbf{wait} : \langle \tau^{1}_{0} \rangle \multimap \tau$$

λ_{lock}'s lock API



$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

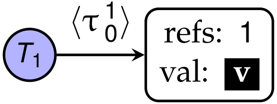
$$\textbf{release} : \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

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

$$\textbf{drop} : \langle \tau^{0}_{0} \rangle \multimap \mathbf{1}$$

$$\textbf{wait} : \langle \tau^{1}_{0} \rangle \multimap \tau$$

λ_{lock}'s lock API



$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

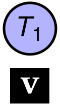
$$\textbf{release} : \langle \tau^{a}_{1} \rangle \times \tau \multimap \langle \tau^{a}_{0} \rangle$$

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

$$\textbf{drop} : \langle \tau^{0}_{0} \rangle \multimap \mathbf{1}$$

$$\textbf{wait} : \langle \tau^{1}_{0} \rangle \multimap \tau$$

λ_{lock}'s lock API



$$\textbf{new} : \mathbf{1} \multimap \langle \tau^{1}_{1} \rangle$$

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

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

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

$$\textbf{drop} : \langle \tau^{0}_{0} \rangle \multimap \mathbf{1}$$

$$\textbf{wait} : \langle \tau^{1}_{0} \rangle \multimap \tau$$

Which concurrency patterns does λ_{lock} support?

Simultaneous acquisition of any locks in scope

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

$\checkmark$

deadlock free

Simultaneous acquisition of any locks in scope

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

✓

deadlock free

Deadlocks?

$$
\begin{aligned}
&\textbf{let}\ x\ =\ \textbf{release}(\textbf{new}(), 1)\ \textbf{in}\\
&\textbf{let}\ y\ =\ \textbf{release}(\textbf{new}(), 2)\ \textbf{in}\\
&\textbf{let}\ x\ =\ \textbf{fork}(x, \lambda x.\ \mathsf{swap}(x, y) \cdots)\ \textbf{in}
\end{aligned}
$$

Variable *y* not in scope here!

Simultaneous acquisition of any locks in scope

$\textbf{let}\ \mathsf{swap}\ =\ \lambda(\ell_1 : \langle\tau_0^{\alpha}\rangle, \ell_2 : \langle\tau_0^{\alpha'}\rangle).$

$\quad\textbf{let}\ (\ell_1 : \langle\tau_1^{\alpha}\rangle, x_1 : \tau)\ =\ \textbf{acquire}(\ell_1)\ \textbf{in}$

$\quad\textbf{let}\ (\ell_2 : \langle\tau_1^{\alpha'}\rangle, x_2 : \tau)\ =\ \textbf{acquire}(\ell_2)\ \textbf{in}$

$\quad\textbf{let}\ \ell_1 : \langle\tau_0^{\alpha}\rangle\ =\ \textbf{release}(\ell_1, x_2)\ \textbf{in}$

$\quad\textbf{let}\ \ell_2 : \langle\tau_0^{\alpha'}\rangle\ =\ \textbf{release}(\ell_2, x_1)\ \textbf{in}$

$\quad(\ell_1, \ell_2)$

✓

deadlock free

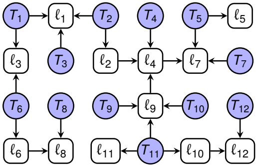
Deadlocks?

$\textbf{let}\ x\ =\ (\cdots)\ \textbf{in}$

$\textbf{let}\ y_1, y_2, y_3, y_4, y_5, y_6\ =\ (\cdots)\ \textbf{in}$

$\textbf{let}\ x\ =\ \textbf{fork}(x, \lambda x.\ \mathsf{foo}(x, y_4, y_5, y_6))\ \textbf{in}$

$\mathsf{bar}(x, y_1, y_2, y_3)$



Storing 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

Futures / promises / fork-join

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

Obligation to fulfill promise cannot be discarded

Other concurrency patterns

Recursive shared mutable data structures

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

Other concurrency patterns

Recursive shared mutable data structures

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

Message passing as a library

With *session types* encoded as λ_{lock} types:

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

Shared & client-server sessions:

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

λ_{lock}'s

Deadlock and Leak Freedom Theorem

Program semantics

$$T_1 \triangleq [\textbf{let}\ x\ =\ \textbf{new}()\ \textbf{in}\ (\cdots)]$$

$$\underbrace{\{T_1\}}_{\rho_1} \rightsquigarrow \underbrace{\{T_1', L_1\}}_{\rho_2} \rightsquigarrow \cdots \rightsquigarrow \underbrace{\{T_1, T_2, ..., L_1, L_2, ...\}}_{\rho_n}$$

Weak deadlock and leak freedom theorem

Program semantics

$$T_1 \triangleq [\textbf{let}\ x\ =\ \textbf{new}()\ \textbf{in}\ (\cdots)]$$

$$\underbrace{\{T_1\}}_{\rho_1} \rightsquigarrow \underbrace{\{T_1', L_1\}}_{\rho_2} \rightsquigarrow \cdots \rightsquigarrow \underbrace{\{T_1, T_2, ..., L_1, L_2, ...\}}_{\rho_n} \overset{?}{\rightsquigarrow} \rho_{n+1}$$

Weak deadlock and leak freedom theorem

Program semantics

$$T_1 \triangleq [\textbf{let } x = \textbf{new}() \textbf{ in } (\cdots)]$$

$$\underbrace{\{T_1\}}_{\rho_1} \rightsquigarrow \underbrace{\{T_1', L_1\}}_{\rho_2} \rightsquigarrow \cdots \rightsquigarrow \underbrace{\{T_1, T_2, ..., L_1, L_2, ...\}}_{\rho_n} \overset{?}{\rightsquigarrow} \rho_{n+1}$$

Global progress

If ρ_1 type-checks and $\rho_1 \rightsquigarrow^* \rho_n \neq \emptyset$, then $\exists \rho_{n+1}.\ \rho_n \rightsquigarrow \rho_{n+1}$

No total deadlocks, leaked memory, run-time type errors, use-after-free, etc.

Program semantics

$$T_1 \triangleq [\textbf{let } x = \textbf{new}()\textbf{ in } (\cdots)]$$

$$\underbrace{\{T_1\}}_{\rho_1} \rightsquigarrow \underbrace{\{T_1', L_1\}}_{\rho_2} \rightsquigarrow \cdots \rightsquigarrow \underbrace{\{T_1, T_2, ..., L_1, L_2, ...\}}_{\rho_n} \overset{?}{\rightsquigarrow} \rho_{n+1}$$

Global progress

If ρ_1 type-checks and $\rho_1 \rightsquigarrow^* \rho_n \neq \emptyset$, then $\exists \rho_{n+1}.\ \rho_n \rightsquigarrow \rho_{n+1}$

No total deadlocks, leaked memory, run-time type errors, use-after-free, etc.

✓ **in Coq** (≈ 13k loc)

Strong deadlock and leak freedom theorem (✓ **in Coq**)

$T \in \rho$ waits for $L \in \rho$ if thread T is blocked on lock L.
$L \in \rho$ waits for $T \in \rho$ if thread T references lock L but is not blocked on L.
$L \in \rho$ waits for $L' \in \rho$ if lock L' references lock L.

Strong deadlock and leak freedom theorem (✓ **in Coq**)

$T \in \rho$ waits for $L \in \rho$ if thread T is blocked on lock L.
$L \in \rho$ waits for $T \in \rho$ if thread T references lock L but is not blocked on L.
$L \in \rho$ waits for $L' \in \rho$ if lock L' references lock L.

$X \in \rho$ is *reachable* if X transitively waits for $T \in \rho$ that can progress

$S \subseteq \rho$ is a *deadlock* if no $T \in S$ can step and no $X \in S$ waits for $Y \in \rho \setminus S$

Strong deadlock and leak freedom theorem (✓ **in Coq**)

$T \in \rho$ <u>waits for</u> $L \in \rho$ if thread T is blocked on lock L.
$L \in \rho$ <u>waits for</u> $T \in \rho$ if thread T references lock L but is not blocked on L.
$L \in \rho$ <u>waits for</u> $L' \in \rho$ if lock L' references lock L.

$X \in \rho$ is *reachable* if X transitively <u>waits for</u> $T \in \rho$ that can progress

$S \subseteq \rho$ is a *deadlock* if no $T \in S$ can step and no $X \in S$ <u>waits for</u> $Y \in \rho \setminus S$

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

Theorem: ρ_1 type checked $\implies$ all $X \in \rho_n$ reachable

Corollary: ρ_1 type checked $\implies$ no deadlocks $\emptyset \subset S \subseteq \rho_n$

Corollary: $\rho_n \neq \emptyset \to \exists \rho_{n+1},\ \rho_n \rightsquigarrow \rho_{n+1}$

Insight: deadlock and leak freedom are related

Deadlock and leak freedom proof sketch

Goal: Find a thread in $\rho_n \neq \emptyset$ that can progress

Fact 1: ρ_1 type-checks $\implies$ **every $X \in \rho_n$ can either progress, or waits for some Y**

Deadlock and leak freedom proof sketch

Goal: Find a thread in $\rho_n \neq \emptyset$ that can progress

Fact 1: ρ_1 type-checks $\implies$ **every $X \in \rho_n$ can either progress, or waits for some Y**

Therefore: Start at any $X \in \rho_n$, and keep following waits for edges

Deadlock and leak freedom proof sketch

Goal: Find a thread in $\rho_n \neq \emptyset$ that can progress

Fact 1: ρ_1 type-checks $\implies$ **every $X \in \rho_n$ can either progress, or waits for some Y**

Therefore: Start at any $X \in \rho_n$, and keep following waits for edges

- **What if we loop in a cycle? deadlock/leak**

Deadlock and leak freedom proof sketch

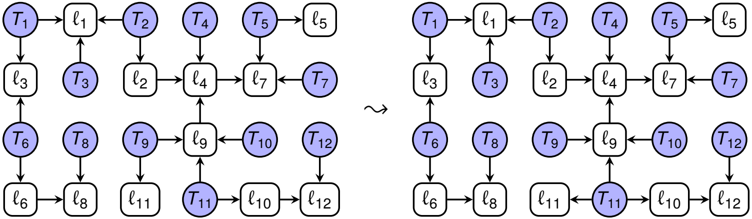
Goal: Find a thread in $\rho_n \neq \emptyset$ that can progress

Fact 1: ρ_1 type-checks $\implies$ **every $X \in \rho_n$ can either progress, or waits for some Y**

Therefore: Start at any $X \in \rho_n$, and keep following waits for edges

- **What if we loop in a cycle? deadlock/leak**

Fact 2: ρ_1 type checks $\implies$ **run-time graph ρ_n of threads & locks remains acyclic**



Deadlock and leak freedom proof sketch

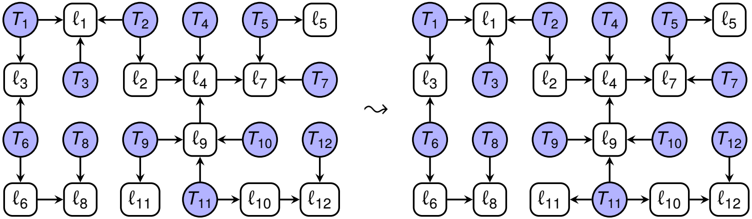
Goal: Find a thread in $\rho_n \neq \emptyset$ that can progress

Fact 1: ρ_1 type-checks $\implies$ **every $X \in \rho_n$ can either progress, or waits for some Y**

Therefore: Start at any $X \in \rho_n$, and keep following waits for edges

- **What if we loop in a cycle? deadlock/leak**

Fact 2: ρ_1 type checks $\implies$ **run-time graph ρ_n of threads & locks remains acyclic**



Problem: **Proving Fact 1 & 2 formally (in Coq) is hard and tedious**

Separation logic for acyclicity

Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic

Jules Jacobs, Stephanie Balzer, Robbert Krebbers, in POPL'22

Motto: **Do proofs in separation logic, get acyclicity for free**

Key insight: Separation logic proof rules correspond to acyclicity-preserving graph transformations!

Motto: **Do proofs in separation logic, get acyclicity for free**

Key insight: Separation logic proof rules correspond to acyclicity-preserving graph transformations!

1. Instantiate Iris proof mode with a linear separation logic
2. Do all type preservation inside that separation logic:

Coq	$\to$	**Iris**
`Prop`	$\to$	`iProp`
$P \wedge Q$	$\to$	$P * Q$
`intros`	$\to$	`iIntros`
`destruct`	$\to$	`iDestruct`
`split`	$\to$	`iSplit`

3. Get acyclicity for free!*

Beyond acyclicity: λ_{lock++}

Beyond acyclicity

Unifying two worlds of session types

Multiparty GV: Functional Multiparty Session Types With Certified Deadlock Freedom
Jules Jacobs, Stephanie Balzer, Robbert Krebbers, in ICFP'22

GV family languages

Gay, Vasconcelos '10, Wadler '12

- Binary session types

MPST family languages

Honda '08

- Multiparty session types

GV family languages
Gay, Vasconcelos ’10, Wadler ’12

- Binary session types
- Deadlock-freedom by duality & linear typing

MPST family languages
Honda ’08

- Multiparty session types
- Deadlock-freedom by global consistency check

GV family languages
Gay, Vasconcelos ’10, Wadler ’12

- Binary session types
- Deadlock-freedom by duality & linear typing
- Dynamic spawning

MPST family languages
Honda ’08

- Multiparty session types
- Deadlock-freedom by global consistency check
- One static session

GV family languages
Gay, Vasconcelos ’10, Wadler ’12

- Binary session types
- Deadlock-freedom by duality & linear typing
- Dynamic spawning
- Channels first class values

MPST family languages
Honda ’08

- Multiparty session types
- Deadlock-freedom by global consistency check
- One static session
- Channels second class

GV family languages
Gay, Vasconcelos '10, Wadler '12

- Binary session types
- Deadlock-freedom by duality & linear typing
- Dynamic spawning
- Channels first class values
- Functional programming

MPST family languages
Honda '08

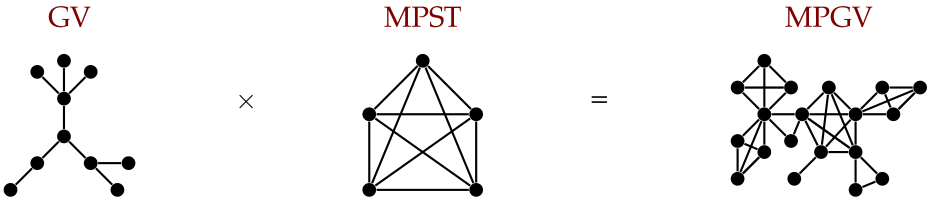
- Multiparty session types
- Deadlock-freedom by global consistency check
- One static session
- Channels second class
- Pi-calculus variants

GV family languages
Gay, Vasconcelos ’10, Wadler ’12

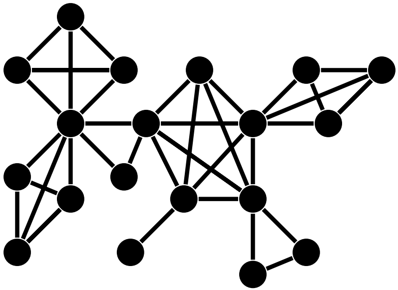
- Binary session types
- Deadlock-freedom by duality & linear typing
- Dynamic spawning
- Channels first class values
- Functional programming

MPST family languages
Honda ’08

- Multiparty session types
- Deadlock-freedom by global consistency check
- One static session
- Channels second class
- Pi-calculus variants



MPGV allows local cycles



Can we allow that in λ_{lock}?

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
  (only for locks in the same lock group)
- We prevent storing locks inside each other
  (only for locks in the same lock group)

$\lambda_{\text{lock++}}$'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

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

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

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

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

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

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

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

$$\textbf{fork} : \langle A\rangle \times (\langle B\rangle \multimap \mathbf{1}) \multimap \langle C\rangle \qquad (\text{where } A = B \oplus C)$$

Swap within a lock group

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

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

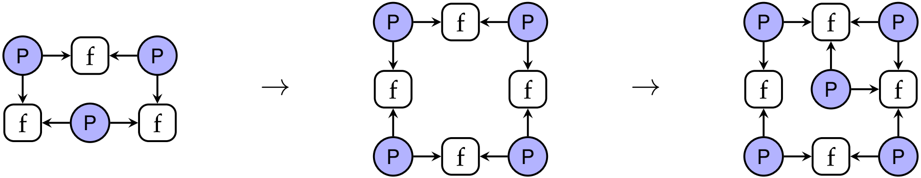
- Type system enforces an order *within* a group
- No restrictions between two groups
  (Partial lock orders don't allow this!)

Dijkstra's dining philosophers

Lock groups allow $\lambda_{\text{lock++}}$ 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
- Only need types of size 3 for table of size n:

$$\langle \mathsf{fork}^a_b, \mathsf{fork}^{a'}_{b'} \rangle \to \langle \mathsf{fork}^a_b, \mathsf{fork}^{a'}_{b'}, \mathsf{fork}^{a''}_{b''} \rangle \to \langle \mathsf{fork}^a_b, \mathsf{fork}^{a''}_{b''} \rangle$$



Conclusion

Rust has a *practical* type system for memory-safety without GC ✓

Rust has a *practical* type system for memory-safety without GC ✓

Can a *practical* type system be deadlock and leak free?

Rust has a *practical* type system for memory-safety without GC ✓

Can a *practical* type system be deadlock and leak free?

I hope to have convinced you that:

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

Rust has a *practical* type system for memory-safety without GC ✓

Can a *practical* type system be deadlock and leak free?

I hope to have convinced you that:

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

Problems yet to be solved:

- Type system → program logic (“Deadlock Free Iris”)
- Integration with Rust features (borrowing & `unsafe`)
- DAG-shaped mutable data structures / `Rc<RefCell<T>>`

> **“The authors didn’t even have to hide a bunch of more complicated rules in an appendix.”** **– Reviewer A**

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)