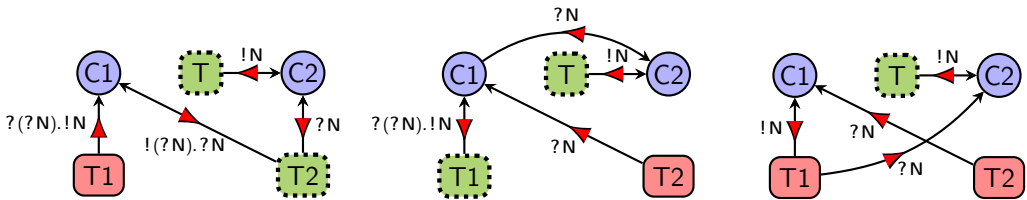
Mechanized Deadlock Freedom for Session Types

Jules Jacobs[1] Robbert Krebbers[1] Stephanie Balzer[2]



[1]Radboud University, The Netherlands
[2]Carnegie Mellon University, USA

Mechanization of session types

State of the art:

- Type safety for higher order binary session types
  (Thiemann 2019, Rouvoet et al. 2020, Hinrichsen et al. 2021)
- Deadlock freedom for a single multiparty session
  (Castro-Perez et al. 2021)

This work:

- Mechanized deadlock and leak freedom for higher order binary session types

Setting: a lambda calculus with session types, inspired by GV

Channel operation	Type signature
`let c = fork(λ c', ...)`	$\texttt{fork} : (s \multimap \mathbf{1}) \multimap \overline{s}$
`let c = send(c,msg)`	$\texttt{send} : (!\,\tau.s) \times \tau \multimap s$
`let (c,msg) = receive(c)`	$\texttt{receive} : ?\,\tau.s \multimap s \times \tau$
`close(c)`	$\texttt{close} : \mathsf{End} \multimap \mathbf{1}$

- Small-step operational semantics with flat thread pool & heap of buffers

Setting: a lambda calculus with session types, inspired by GV

Channel operation	Type signature
`let c = fork(λ c', ...)`	$\texttt{fork} : (s \multimap \mathbf{1}) \multimap \overline{s}$
`let c = send(c,msg)`	$\texttt{send} : (!\,\tau.s) \times \tau \multimap s$
`let (c,msg) = receive(c)`	$\texttt{receive} : ?\,\tau.s \multimap s \times \tau$
`close(c)`	$\texttt{close} : \mathsf{End} \multimap \mathbf{1}$

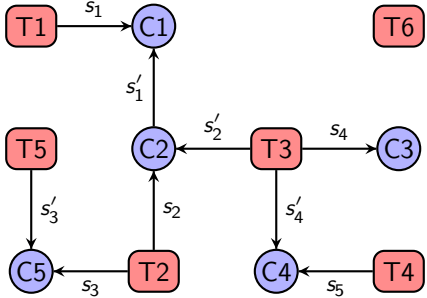
- Small-step operational semantics with flat thread pool & heap of buffers
- Untyped programs can **deadlock** (e.g. due to cyclic waiting dependency)
- Untyped programs can **leak memory** (e.g. due to reference cycles)
- **Our goal:** Mechanized proof that typed programs don't deadlock & don't leak

Setting: a lambda calculus with session types, inspired by GV

Channel operation	Type signature
`let c = fork(λ c', ...)`	$\texttt{fork} : (s \multimap \mathbf{1}) \multimap \overline{s}$
`let c = send(c,msg)`	$\texttt{send} : (!\,\tau.s) \times \tau \multimap s$
`let (c,msg) = receive(c)`	$\texttt{receive} : ?\,\tau.s \multimap s \times \tau$
`close(c)`	$\texttt{close} : \mathsf{End} \multimap \mathbf{1}$

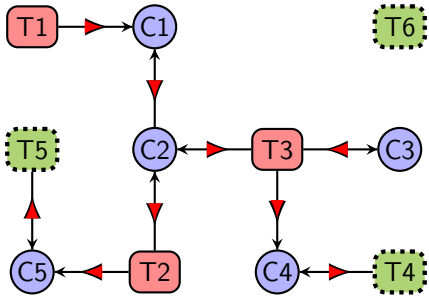
- Small-step operational semantics with flat thread pool & heap of buffers
- Untyped programs can **deadlock** (e.g. due to cyclic waiting dependency)
- Untyped programs can **leak memory** (e.g. due to reference cycles)
- **Our goal:** Mechanized proof that typed programs don't deadlock & don't leak
- **Probem:** reasoning about dependency structure in a proof assistant is hard
- **Our approach:** develop ***connectivity graph*** framework

Connectivity graphs

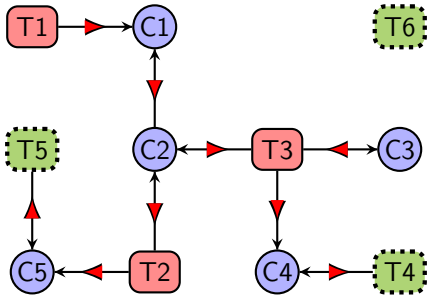


- Vertices represent threads & channels
- Edges represent references, labeled with the session type
- Keeps track of heap typing and reference topology simultaneously
- Progress & preservation style proof with the following invariant:
  - The configuration has an *acyclic* connectivity graph
  - Each thread & channel satisfies a *local invariant* linking its configuration state with the session types on its edges in the graph

Waiting induction principle



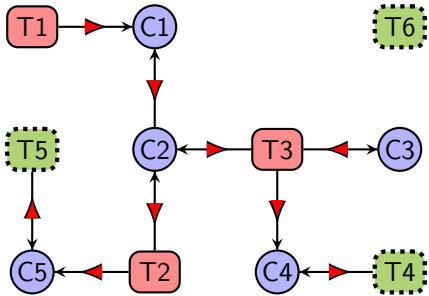
Waiting induction principle



Lemma (Waiting induction)

To prove $P(v)$, we may assume $P(w)$ for all $w \blacktriangleleft v$.

Waiting induction principle

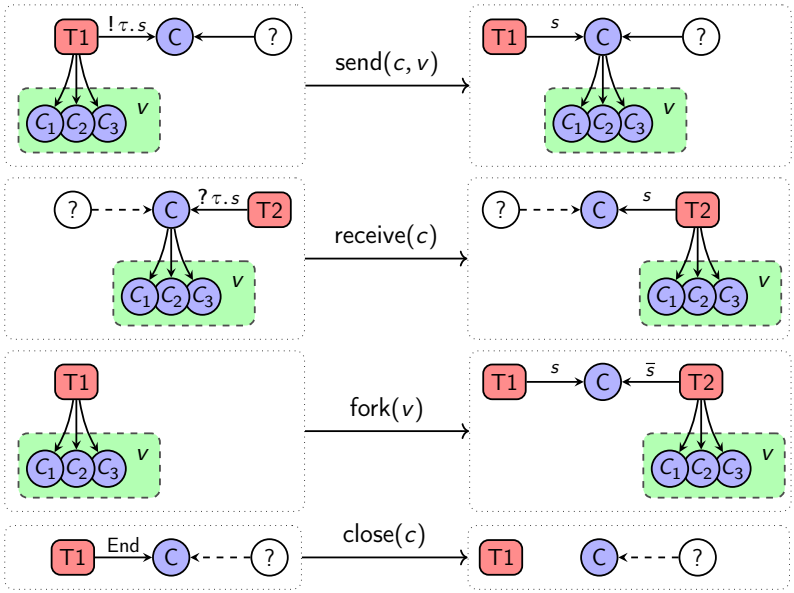


Lemma (Waiting induction)

To prove $P(v)$, we may assume $P(w)$ for all $w \blacktriangleleft v$.

- Used to prove invariant $\Longrightarrow$ deadlock freedom
- This deadlock freedom proof does only local, language specific reasoning.
- **Graph acyclicity reasoning is encapsulated in *generic* waiting induction.**

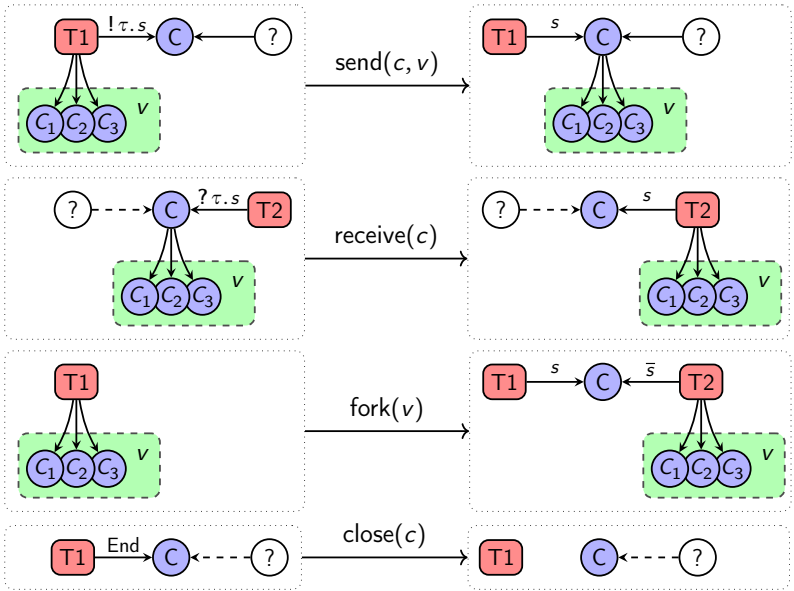
Local graph transformations



Preserves:

- Invariants
- Acyclicity

Local graph transformations



Preserves:

- Invariants
- Acyclicity

Generic?

Separation logic local invariants

We state the local invariant for each vertex in separation logic:

- Local invariant links a vertex to its run-time configuration state
- Local invariant can talk about incoming edges and outgoing edges
- Outgoing edges become separation logic resources

Separation logic local invariants

We state the local invariant for each vertex in separation logic:

- Local invariant links a vertex to its run-time configuration state
- Local invariant can talk about incoming edges and outgoing edges
- Outgoing edges become separation logic resources

Local invariant for threads:

- The expression is well-typed in the *run-time type system* $\Gamma; \Sigma \vdash e : \tau$
- Σ-environment maps channel references to session types, and is given by the *outgoing* edges of the thread's vertex
- We keep Σ implicit by using separation logic: $(\Gamma \vDash e : \tau) \in \mathit{iProp}$ (inspired by Rouvoet et al.'s approach for typed interpreters)

Separation logic local invariants

We state the local invariant for each vertex in separation logic:

- Local invariant links a vertex to its run-time configuration state
- Local invariant can talk about incoming edges and outgoing edges
- Outgoing edges become separation logic resources

Local invariant for threads:

- The expression is well-typed in the *run-time type system* $\Gamma; \Sigma \vdash e : \tau$
- Σ-environment maps channel references to session types, and is given by the *outgoing* edges of the thread's vertex
- We keep Σ implicit by using separation logic: $(\Gamma \vDash e : \tau) \in \mathit{iProp}$ (inspired by Rouvoet et al.'s approach for typed interpreters)

Local invariant for channels:

- The buffers are consistent with the session types on the *incoming* edges
- The values in the buffers are well typed with respect to the *outgoing* edges

$$\frac{\Gamma = \{x \mapsto \tau\}}{\Gamma \vdash x : \tau} \qquad \frac{\cdot}{\emptyset \vdash () : \mathbf{1}} \qquad \frac{n \in \mathbb{N}}{\emptyset \vdash n : \mathbf{N}} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \uplus \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{N} \qquad \Gamma_2 \vdash e_2 : \tau \qquad \Gamma_2 \vdash e_3 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

$$\frac{\Gamma \vdash e : \overline{s} \multimap \mathbf{1}}{\Gamma \vdash \mathsf{fork}(e) : s} \qquad \frac{\Gamma_1 \vdash e_1 : !\,\tau.s \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{send}(e_1, e_2) : s} \qquad \frac{\Gamma \vdash e : ?\,\tau.s}{\Gamma \vdash \mathsf{receive}(e) : s \times \tau} \qquad \frac{\Gamma \vdash e : \mathsf{End}}{\Gamma \vdash \mathsf{close}(e) : \mathbf{1}}$$

$$\frac{\ulcorner \Gamma = \{x \mapsto \tau\} \urcorner}{\Gamma \vDash x : \tau}* \qquad \frac{}{\emptyset \vDash () : \mathbf{1}}* \qquad \frac{\ulcorner n \in \mathbb{N} \urcorner}{\emptyset \vDash n : \mathbf{N}}* \qquad \frac{\Gamma_1 \vDash e_1 : \tau_1 \ * \ \Gamma_2 \vDash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vDash (e_1, e_2) : \tau_1 \times \tau_2}*$$

$$\frac{\Gamma \uplus \{x \mapsto \tau_1\} \vDash e : \tau_2}{\Gamma \vDash \lambda x.\ e : \tau_1 \multimap \tau_2}* \qquad \frac{\Gamma_1 \vDash e_1 : \tau_1 \multimap \tau_2 \ * \ \Gamma_2 \vDash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vDash e_1 \ e_2 : \tau_2}*$$

$$\frac{\Gamma_1 \vDash e_1 : \tau_1 \ * \ \Gamma_2 \uplus \{x \mapsto \tau_1\} \vDash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vDash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}* \qquad \frac{\Gamma_1 \vDash e_1 : \mathbf{N} \ * \ (\Gamma_2 \vDash e_2 : \tau \quad \wedge \quad \Gamma_2 \vDash e_3 : \tau)}{\Gamma_1 \uplus \Gamma_2 \vDash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}*$$

$$\frac{\Gamma \vDash e : \overline{s} \multimap 1}{\Gamma \vDash \mathsf{fork}(e) : s}* \qquad \frac{\Gamma_1 \vDash e_1 : !\,\tau.s \ * \ \Gamma_2 \vDash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vDash \mathsf{send}(e_1, e_2) : s}* \qquad \frac{\Gamma \vDash e : ?\,\tau.s}{\Gamma \vDash \mathsf{receive}(e) : s \times \tau}*$$

$$\frac{\Gamma \vDash e : \mathsf{End}}{\Gamma \vDash \mathsf{close}(e) : \mathbf{1}}* \qquad \frac{\mathsf{own}(\mathsf{Chan}(a) \mapsto (t, s))}{\emptyset \vDash \#a_t : s}*$$

Advantages of separation logic

- Separation logic is usually used for verifying individual programs
- But also for type safety proofs using logical relations (e.g. Iris), and definitional interpreters (Rouvoet et al.)
- We use it in a progress & preservation style proof

Advantages of separation logic

- Separation logic is usually used for verifying individual programs
- But also for type safety proofs using logical relations (e.g. Iris), and definitional interpreters (Rouvoet et al.)
- We use it in a progress & preservation style proof

Advantages of separation logic for mechanized linear type systems:

- Automatically takes care of zillions of formal disjointness conditions
- Heap typing Σ only shows up when relevant, and is completely hidden otherwise
- Provides intuitive high-level *ownership reasoning*, even for syntactic properties

Advantages of separation logic

- Separation logic is usually used for verifying individual programs
- But also for type safety proofs using logical relations (e.g. Iris), and definitional interpreters (Rouvoet et al.)
- We use it in a progress & preservation style proof

Advantages of separation logic for mechanized linear type systems:

- Automatically takes care of zillions of formal disjointness conditions
- Heap typing Σ only shows up when relevant, and is completely hidden otherwise
- Provides intuitive high-level *ownership reasoning*, even for syntactic properties

$$(K[e] : B) \iff \exists A.\ (e : A) \wedge \forall e'.\ (e' : A) \to (K[e'] : B) \qquad \text{(traditional lemma)}$$

$$(K[e] : B) \dashv\vdash \exists A.\ (e : A) * \forall e'.\ (e' : A) \mathrel{-\!\!*} (K[e'] : B) \qquad \text{(linear+heap lemma)}$$

Advantages of separation logic

- Separation logic is usually used for verifying individual programs
- But also for type safety proofs using logical relations (e.g. Iris), and definitional interpreters (Rouvoet et al.)
- We use it in a progress & preservation style proof

Advantages of separation logic for mechanized linear type systems:

- Automatically takes care of zillions of formal disjointness conditions
- Heap typing Σ only shows up when relevant, and is completely hidden otherwise
- Provides intuitive high-level *ownership reasoning*, even for syntactic properties

$$(K[e] : B) \iff \exists A.\ (e : A) \wedge \forall e'.\ (e' : A) \to (K[e'] : B) \qquad \text{(traditional lemma)}$$

$$(K[e] : B) \dashv\vdash \exists A.\ (e : A) * \forall e'.\ (e' : A) \mathrel{-\!\!*} (K[e'] : B) \qquad \text{(linear+heap lemma)}$$

Without separation logic:

$$(\Sigma \vdash K[e] : B) \iff \exists A, \Sigma_1, \Sigma_2.\ (\Sigma_1 \cap \Sigma_2 = \emptyset \wedge \Sigma = \Sigma_1 \cup \Sigma_2) \wedge (\Sigma_1 \vdash e : A) \wedge \forall e', \Sigma_3.\ (\Sigma_2 \cap \Sigma_3 = \emptyset \wedge \Sigma_2 \vdash e' : A) \to (\Sigma_2 \cup \Sigma_3 \vdash K[e'] : B)$$

Graph transformations in separation logic

Lemmas for maintaining the invariant when adding, removing, and relabeling edges, and **exchanging** separation logic resources.

Graph transformations in separation logic

Lemmas for maintaining the invariant when adding, removing, and relabeling edges, and **exchanging** separation logic resources.

Lemma (Exchange)

Let $v_1, v_2 \in V$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*

1. $P(v, \Delta) \mathrel{-\!\!*} P'(v, \Delta)$ *for all* $v \in V \setminus \{v_1, v_2\}$ *and* $\Delta \in \mathit{Multiset}\ L$
2. $P(v_1, \Delta_1) \mathrel{-\!\!*} \exists l.\, \mathsf{own}(v_2 \mapsto l) * \forall \Delta_2 \in \mathit{Multiset}\ L.\, P(v_2, \{l\} \uplus \Delta_2) \mathrel{-\!\!*} \exists l'.\, (\mathsf{own}(v_2 \mapsto l') \mathrel{-\!\!*} P'(v_1, \Delta_1)) * P'(v_2, \{l'\} \uplus \Delta_2)$
   for all $\Delta_1 \in \mathit{Multiset}\ L$

Graph transformations in separation logic

Lemmas for maintaining the invariant when adding, removing, and relabeling edges, and **exchanging** separation logic resources.

Lemma (Exchange)

Let $v_1, v_2 \in V$. *To prove* $\mathrm{wf}(P)$ *implies* $\mathrm{wf}(P')$, *it suffices to prove:*

1. $P(v, \Delta) \mathrel{-\!\!*} P'(v, \Delta)$ *for all* $v \in V \setminus \{v_1, v_2\}$ *and* $\Delta \in \mathit{Multiset}\ L$
2. $P(v_1, \Delta_1) \mathrel{-\!\!*} \exists l.\, \mathrm{own}(v_2 \mapsto l) * \forall \Delta_2 \in \mathit{Multiset}\ L.\, P(v_2, \{l\} \uplus \Delta_2) \mathrel{-\!\!*} \exists l'.\, (\mathrm{own}(v_2 \mapsto l') \mathrel{-\!\!*} P'(v_1, \Delta_1)) * P'(v_2, \{l'\} \uplus \Delta_2)$
   for all $\Delta_1 \in \mathit{Multiset}\ L$

Preservation proof appears to do no graph reasoning at all!

- The construction of the new connectivity graph, and the proof of its acyclicity, is encapsulated in the *generic* lemmas.
- The preservation proof does only local, language specific reasoning.

Mechanization

Our language:

1. Functional language + session-typed channels
2. Linear and unrestricted types
   - Unrestricted: numbers, sums, products, unrestricted function type ($\to$)
   - Linear: channels, sums, products, linear function type ($\multimap$)
3. General recursive types: coinductive method adapted from Gay et al. [2020]
   - Recursive session types, including through the message
   - Algebraic data types using recursion + sums + products

Mechanization

Our language:

1. Functional language + session-typed channels
2. Linear and unrestricted types
   - Unrestricted: numbers, sums, products, unrestricted function type ($\rightarrow$)
   - Linear: channels, sums, products, linear function type ($\multimap$)
3. General recursive types: coinductive method adapted from Gay et al. [2020]
   - Recursive session types, including through the message
   - Algebraic data types using recursion + sums + products

Mechanization in Coq:

- Generic $Cgraph(V, L)$ library: 4999 LOC
- Language definition: 451 LOC
- Language specific and leak freedom proof: 1688 LOC

Mechanization

Our language:

1. Functional language + session-typed channels
2. Linear and unrestricted types
   - Unrestricted: numbers, sums, products, unrestricted function type ($\rightarrow$)
   - Linear: channels, sums, products, linear function type ($\multimap$)
3. General recursive types: coinductive method adapted from Gay et al. [2020]
   - Recursive session types, including through the message
   - Algebraic data types using recursion + sums + products

Mechanization in Coq:

- Generic *Cgraph*(V, L) library: 4999 LOC
- Language definition: 451 LOC
- Language specific and leak freedom proof: 1688 LOC

Initial direct attempt: proofs goals got too complex.
Graph reasoning intertwined with language specifics.
Encapsulating the graph reasoning made it manageable.

Questions?

julesjacobs@gmail.com