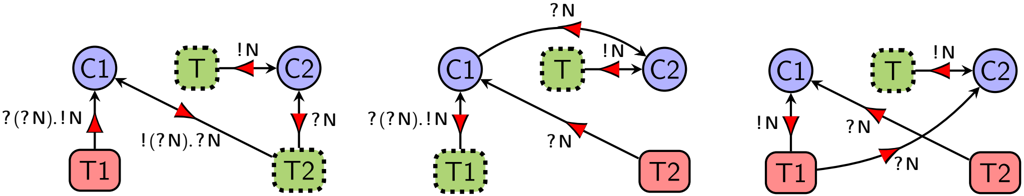
Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic

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



[1]Radboud University Nijmegen
[2]Carnegie Mellon University

POPL'22/Iris workshop'22

Safety in Iris

Iris safety theorem: **all threads can always step**

Safety in Iris

Iris safety theorem: **all threads can always step**

- Spinlock CAS loops can always step...but **deadlock not ruled out**

Safety in Iris

Iris safety theorem: **all threads can always step**

- Spinlock CAS loops can always step...but **deadlock not ruled out**

We wanted: **deadlock-free Iris**

Safety in Iris

Iris safety theorem: **all threads can always step**

- Spinlock CAS loops can always step...but **deadlock not ruled out**

We wanted: **deadlock-free Iris**
We got: progress & preservation style proof of deadlock freedom for session types

- Uses separation logic
- **Maybe the techniques help toward deadlock free Iris**

Session types

Message passing concurrency with first-class channels (Honda [1993])

$$c\ :\ !\mathit{Nat}.\,?\mathit{Bool}.\,!(?\mathit{String}.\,!\mathit{Nat}.\,\mathrm{End}).\,\mathrm{End}$$

Session types

Message passing concurrency with first-class channels (Honda [1993])

$$c\ : !Nat.?Bool.!(?String.!Nat.\mathrm{End}).\mathrm{End}$$

$$\Updownarrow \text{ dual}$$

$$c' : ?Nat.!Bool.?(?String.!Nat.\mathrm{End}).\mathrm{End}$$

Session types

Message passing concurrency with first-class channels (Honda [1993])

$$c\ :\ !Nat.\,?Bool.\,!(?String.\,!Nat.\,\mathrm{End}).\,\mathrm{End}$$

$$\Updownarrow \text{dual}$$

$$c' : ?Nat.\,!Bool.\,?(?String.\,!Nat.\,\mathrm{End}).\,\mathrm{End}$$

GV: functional programming with session types
(Gay and Vasconcelos [2010], Wadler [2012])

$$\mathrm{fork} : (s \multimap 1) \to \overline{s} \qquad \mathrm{send} : (!t.\,s) \times t \multimap s$$

$$\mathrm{close} : \mathrm{End} \multimap 1 \qquad \mathrm{receive} : ?t.\,s \multimap s \times t$$

$$\mathrm{let}\ c = \mathrm{fork}(\lambda c'.\ \ldots\mathrm{receive}(c')\ldots)\ \mathrm{in}\ \mathrm{send}(c, 23)\ldots$$

What makes session types interesting

Linear session types: cannot copy or delete a channel reference before you are done

What makes session types interesting

Linear session types: cannot copy or delete a channel reference before you are done

- Required for **type safety**; **mechanized with Actris by Hinrichsen et al. [2020, 2021]** (and by Castro-Perez et al. [2020], Ciccone and Padovani [2020], Goto et al. [2016], Rouvoet et al. [2020], Thiemann [2019], ...)

What makes session types interesting

Linear session types: cannot copy or delete a channel reference before you are done

- Required for **type safety**; **mechanized with Actris by Hinrichsen et al. [2020, 2021]** (and by Castro-Perez et al. [2020], Ciccone and Padovani [2020], Goto et al. [2016], Rouvoet et al. [2020], Thiemann [2019], ...)
- But also guarantees **deadlock freedom**, **global progress**; **proved by Carbone et al. [2008, 2010]** (and Caires and Pfenning [2010], Lindley and Morris [2015], Wadler [2012],...)

  Not yet mechanized: requires reasoning about graphs

What makes session types interesting

Linear session types: cannot copy or delete a channel reference before you are done

- Required for **type safety**; **mechanized with Actris by Hinrichsen et al. [2020, 2021]** (and by Castro-Perez et al. [2020], Ciccone and Padovani [2020], Goto et al. [2016], Rouvoet et al. [2020], Thiemann [2019], ...)
- But also guarantees **deadlock freedom**, **global progress**; **proved by Carbone et al. [2008, 2010]** (and Caires and Pfenning [2010], Lindley and Morris [2015], Wadler [2012],...)

  Not yet mechanized: requires reasoning about graphs

Why session types give deadlock freedom

Two owners per channel

- Duality of channel types → no simple deadlocks
- Linear typing maintains acyclicity of ownership structure → no cyclic deadlocks

Why session types give deadlock freedom

Two owners per channel

- Duality of channel types → no simple deadlocks
- Linear typing maintains acyclicity of ownership structure → no cyclic deadlocks

Even when channels are asynchronous and first-class values:

- dynamically created
- sent as messages over channels
- stored in data structures
- captured by closures
- in Turing-complete language (→ termination argument doesn't work)

Why session types give deadlock freedom

Two owners per channel

- Duality of channel types → no simple deadlocks
- Linear typing maintains acyclicity of ownership structure → no cyclic deadlocks

Even when channels are asynchronous and first-class values:

- dynamically created
- sent as messages over channels
- stored in data structures
- captured by closures
- in Turing-complete language (→ termination argument doesn't work)

Initial direct attempt: proofs goals got too complex.
Graph reasoning intertwined with language specifics.

Why session types give deadlock freedom

Two owners per channel

- Duality of channel types → no simple deadlocks
- Linear typing maintains acyclicity of ownership structure → no cyclic deadlocks

Even when channels are asynchronous and first-class values:

- dynamically created
- sent as messages over channels
- stored in data structures
- captured by closures
- in Turing-complete language (→ termination argument doesn't work)

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

Contribution: connectivity graph proof method

This work: connectivity graphs

- Method for factoring out graph reasoning from reasoning about typing
- Mechanized in the Coq proof assistant
- Applied to prove deadlock freedom for feature-rich session-typed language
- Abstract representation of run-time configuration

Contribution: connectivity graph proof method

This work: connectivity graphs

- Method for factoring out graph reasoning from reasoning about typing
- Mechanized in the Coq proof assistant
- Applied to prove deadlock freedom for feature-rich session-typed language
- Abstract representation of run-time configuration

Run-time configuration ρ

Threads: $\{T_1 \mapsto e_1, ..., T_6 \mapsto e_6\}$
Channels: $\{C_1 \mapsto \text{buf}_1, ..., C_5 \mapsto \text{buf}_5\}$

Contribution: connectivity graph proof method

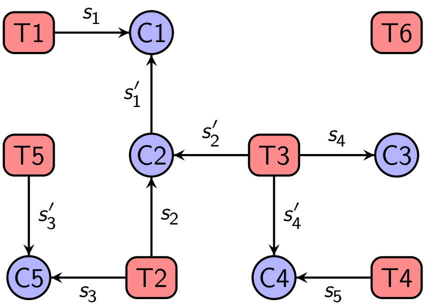
This work: connectivity graphs

- Method for factoring out graph reasoning from reasoning about typing
- Mechanized in the Coq proof assistant
- Applied to prove deadlock freedom for feature-rich session-typed language
- Abstract representation of run-time configuration

Run-time configuration ρ

Threads: $\{T_1 \mapsto e_1, ..., T_6 \mapsto e_6\}$
Channels: $\{C_1 \mapsto \text{buf}_1, ..., C_5 \mapsto \text{buf}_5\}$

Connectivity graph G



Contribution: connectivity graph proof method

This work: connectivity graphs

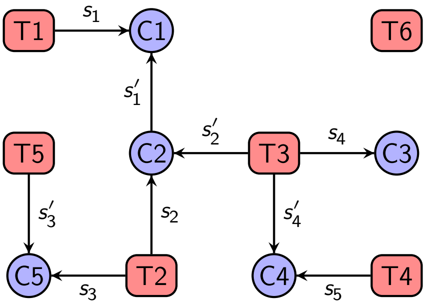
- Method for factoring out graph reasoning from reasoning about typing
- Mechanized in the Coq proof assistant
- Applied to prove deadlock freedom for feature-rich session-typed language
- Abstract representation of run-time configuration

Run-time configuration ρ

Threads: $\{T_1 \mapsto e_1, ..., T_6 \mapsto e_6\}$
Channels: $\{C_1 \mapsto \mathrm{buf}_1, ..., C_5 \mapsto \mathrm{buf}_5\}$

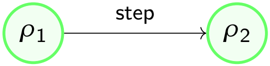
$$G \vDash \rho$$
$$wf(\rho) := \exists G. G \vDash \rho$$

Connectivity graph G

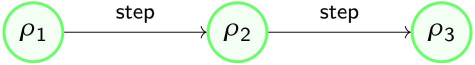


Connectivity graph proof based on progress and preservation

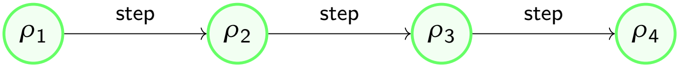
Connectivity graph proof based on progress and preservation



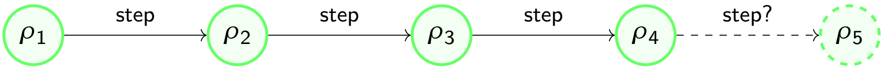
Connectivity graph proof based on progress and preservation



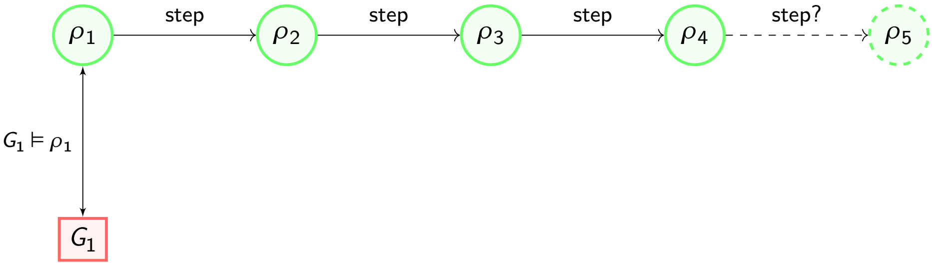
Connectivity graph proof based on progress and preservation



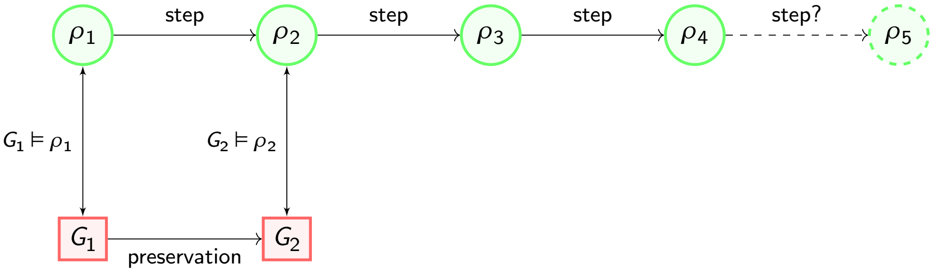
Connectivity graph proof based on progress and preservation



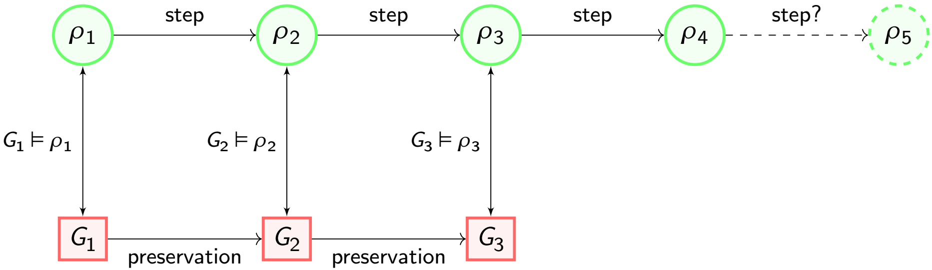
Connectivity graph proof based on progress and preservation



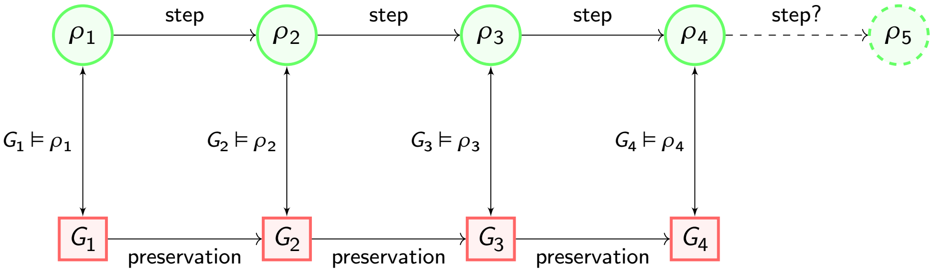
Connectivity graph proof based on progress and preservation



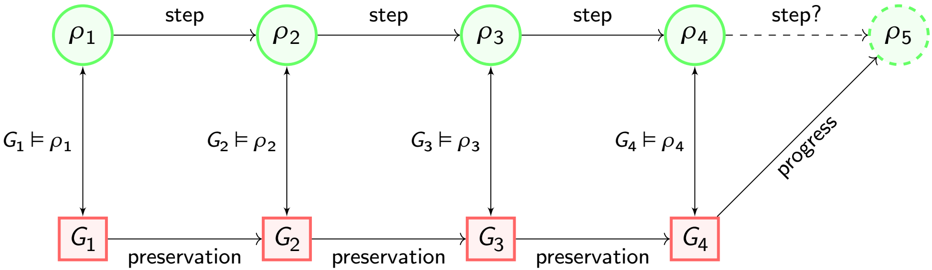
Connectivity graph proof based on progress and preservation



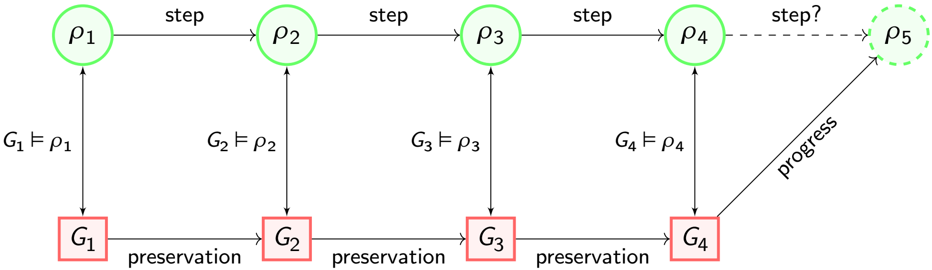
Connectivity graph proof based on progress and preservation



Connectivity graph proof based on progress and preservation



Connectivity graph proof based on progress and preservation



Connectivity graph framework:

- $Cgraph(V, L)$ data type for acyclic labeled graphs
- Generic construction for $wf(\rho) := \overline{wf}(P_\rho)$
  - Parameterized by local separation logic predicate $P_\rho(v)$ for each vertex $v \in G$
- Preservation: graph transformations in separation logic
- Progress: waiting induction principle for $Cgraph(V, L)$

All generic over vertices V and labels L

Linear heap typing in separation logic: (cf. Rouvoet [2020]'s definitional interpreters)

$$\frac{\Sigma_1 \vdash e_1 : \tau_1 \qquad \Sigma_2 \vdash e_2 : \tau_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1 \cup \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

Linear heap typing in separation logic: (cf. Rouvoet [2020]'s definitional interpreters)

$$\frac{\Sigma_1 \vdash e_1 : \tau_1 \qquad \Sigma_2 \vdash e_2 : \tau_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1 \cup \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \Rightarrow \quad \frac{e_1 : \tau_1 \; * \; e_2 : \tau_2}{(e_1, e_2) : \tau_1 \times \tau_2}*$$

Linear heap typing in separation logic: (cf. Rouvoet [2020]'s definitional interpreters)

$$\frac{\Sigma_1 \vdash e_1 : \tau_1 \qquad \Sigma_2 \vdash e_2 : \tau_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1 \cup \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \Rightarrow \quad \frac{e_1 : \tau_1 \;*\; e_2 : \tau_2}{(e_1, e_2) : \tau_1 \times \tau_2}*$$

$$\frac{\Sigma = \{\mathsf{Chan}(a) \mapsto (t, s)\}}{\Sigma \vdash \#a_t : s} \quad \Rightarrow \quad \frac{\mathsf{own}(\mathsf{Chan}(a) \mapsto (t, s))}{\#a_t : s}*$$

Linear heap typing in separation logic: (cf. Rouvoet [2020]'s definitional interpreters)

$$\frac{\Sigma_1 \vdash e_1 : \tau_1 \qquad \Sigma_2 \vdash e_2 : \tau_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1 \cup \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \Rightarrow \quad \frac{e_1 : \tau_1 \ * \ e_2 : \tau_2}{(e_1, e_2) : \tau_1 \times \tau_2}*$$

$$\frac{\Sigma = \{\mathrm{Chan}(a) \mapsto (t, s)\}}{\Sigma \vdash \#a_t : s} \quad \Rightarrow \quad \frac{\mathrm{own}(\mathrm{Chan}(a) \mapsto (t, s))}{\#a_t : s}*$$

For vertex v in the graph, separation logic resource $\Sigma = \mathrm{OutEdges}(v)$

Linear heap typing in separation logic: (cf. Rouvoet [2020]'s definitional interpreters)

$$\frac{\Sigma_1 \vdash e_1 : \tau_1 \qquad \Sigma_2 \vdash e_2 : \tau_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1 \cup \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \Rightarrow \quad \frac{e_1 : \tau_1 \;\ast\; e_2 : \tau_2}{(e_1, e_2) : \tau_1 \times \tau_2}\ast$$

$$\frac{\Sigma = \{\mathsf{Chan}(a) \mapsto (t, s)\}}{\Sigma \vdash \#a_t : s} \quad \Rightarrow \quad \frac{\mathsf{own}(\mathsf{Chan}(a) \mapsto (t, s))}{\#a_t : s}\ast$$

For vertex v in the graph, separation logic resource $\Sigma = \mathsf{OutEdges}(v)$

Lemmas in separation logic:

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

Linear heap typing in separation logic: (cf. Rouvoet [2020]'s definitional interpreters)

$$\frac{\Sigma_1 \vdash e_1 : \tau_1 \qquad \Sigma_2 \vdash e_2 : \tau_2 \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{\Sigma_1 \cup \Sigma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \Rightarrow \quad \frac{e_1 : \tau_1 \;\; * \;\; e_2 : \tau_2}{(e_1, e_2) : \tau_1 \times \tau_2} *$$

$$\frac{\Sigma = \{\mathrm{Chan}(a) \mapsto (t, s)\}}{\Sigma \vdash \#a_t : s} \quad \Rightarrow \quad \frac{\mathrm{own}(\mathrm{Chan}(a) \mapsto (t, s))}{\#a_t : s} *$$

For vertex v in the graph, separation logic resource $\Sigma = \mathrm{OutEdges}(v)$

Lemmas in separation logic:

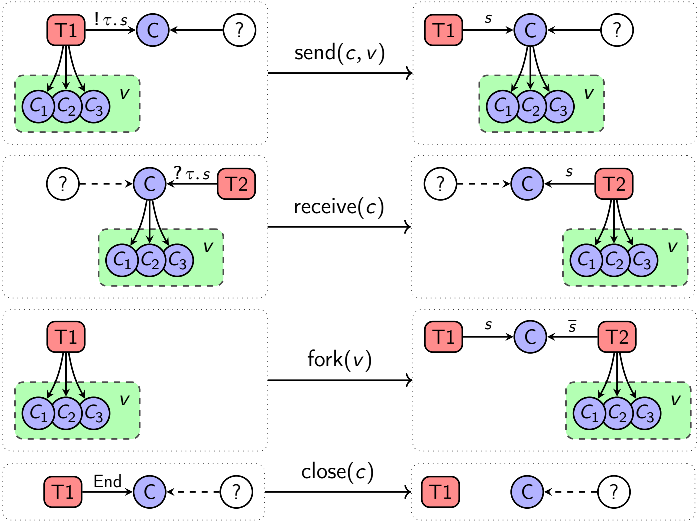
$$\begin{aligned}(\Sigma \vdash K[e] : B) \iff \exists A, \Sigma_1, \Sigma_2.\ (\Sigma_1 \cap \Sigma_2 = \emptyset) \wedge (\Sigma = \Sigma_1 \cup \Sigma_2) \wedge (\Sigma_1 \vdash e : A) \wedge \\ \forall e', \Sigma_3.\ (\Sigma_2 \cap \Sigma_3 = \emptyset) \wedge (\Sigma_2 \vdash e' : A) \to (\Sigma_2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

$$\Rightarrow$$

$$(K[e] : B) \quad \dashv\vdash \quad \exists A.\ (e : A) * \forall e'.\ (e' : A) \mathrel{-\!\!*} (K[e'] : B)$$

We use the Iris proof mode to reason in separation logic (Krebbers et al. [2017])

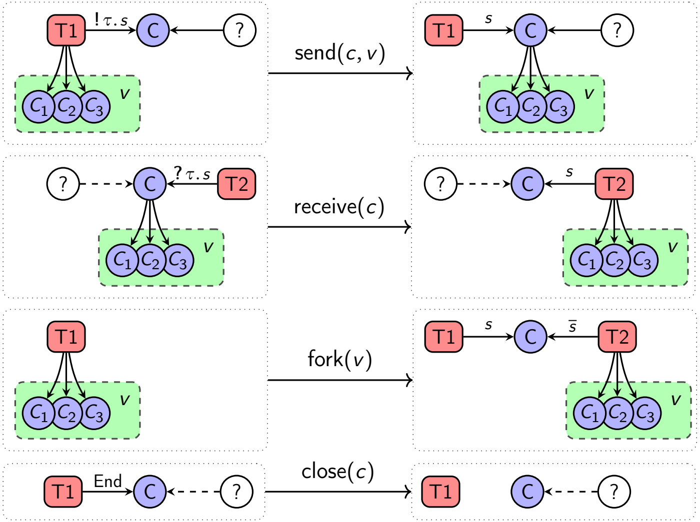
Preservation via local graph transformations



Preserves:

- Acyclicity
- Local predicates $P_\rho(v)$ used for $\overline{wf}(P_\rho)$

Preservation via local graph transformations



Preserves:

- Acyclicity
- Local predicates $P_\rho(v)$ used for $\overline{wf}(P_\rho)$

In separation logic: if

$$P_\rho(T_1) * (\mathrm{own}(C \mapsto s) -\!\!* P_\rho(C))$$

$$\vdash$$

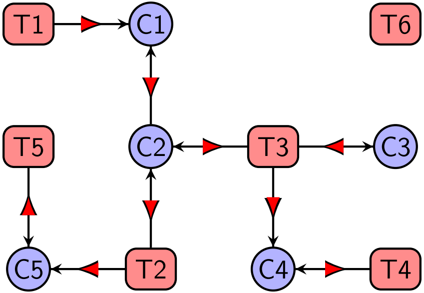
$$(\mathrm{own}(C \mapsto s') -\!\!* P_{\rho'}(T_1)) * P_{\rho'}(C)$$

then: $\overline{wf}(P_\rho) \rightarrow \overline{wf}(P_{\rho'})$

Explained in our paper!

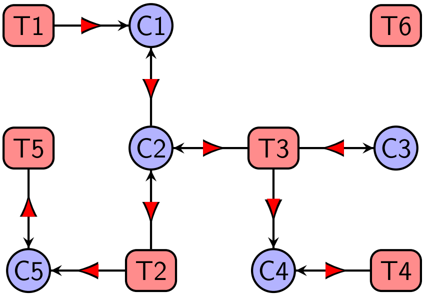
Progress via waiting induction

Connectivity graph with *waiting dependencies* (►)
derived from run-time configuration



Progress via waiting induction

Connectivity graph with *waiting dependencies* (►)
derived from run-time configuration



Lemma (Waiting induction)

Let $R(v, w)$ be any relation on the vertices. To prove $P(v)$, we may assume $P(w)$ for all w such that $v \longrightarrow w$ and $R(v, w)$, or $w \longrightarrow v$ and $\neg R(w, v)$

Mechanization

Mechanization in Coq:

- Generic *Cgraph*(V, L) library: 4999 LOC
- Channels + unrestricted & recursive types language definition: 451 LOC
- Language specific deadlock and leak freedom proof: 1688 LOC

μGV: linear λ-calculus + fork with single-shot atomic exchange

- Global progress & deadlock freedom in Coq (1478 LOC)

Mechanization

Mechanization in Coq:

- Generic *Cgraph*(V, L) library: 4999 LOC
- Channels + unrestricted & recursive types language definition: 451 LOC
- Language specific deadlock and leak freedom proof: 1688 LOC

μGV: linear λ-calculus + fork with single-shot atomic exchange

- Global progress & deadlock freedom in Coq (1478 LOC)

MPGV: linear λ-calculus with multiparty session types

- Global progress & deadlock freedom in Coq (10400 LOC)

`https://github.com/julesjacobs/cgraphs`

Lessons learnt for deadlock-free Iris

- Benchmark: deadlock freedom for session types using *semantic typing* with Actris

Lessons learnt for deadlock-free Iris

- Benchmark: deadlock freedom for session types using *semantic typing* with Actris
- In general, *memory leak freedom* and *deadlock freedom* are equally hard (thanks to POPL reviewers)

Lessons learnt for deadlock-free Iris

- Benchmark: deadlock freedom for session types using *semantic typing* with Actris
- In general, *memory leak freedom* and *deadlock freedom* are equally hard (thanks to POPL reviewers)
- Graph acyclicity is key for higher-order deadlock freedom

Lessons learnt for deadlock-free Iris

- Benchmark: deadlock freedom for session types using *semantic typing* with Actris
- In general, *memory leak freedom* and *deadlock freedom* are equally hard (thanks to POPL reviewers)
- Graph acyclicity is key for higher-order deadlock freedom
- Separation logic is well-matched with graph acyclicity

Lessons learnt for deadlock-free Iris

- Benchmark: deadlock freedom for session types using *semantic typing* with Actris
- In general, *memory leak freedom* and *deadlock freedom* are equally hard (thanks to POPL reviewers)
- Graph acyclicity is key for higher-order deadlock freedom
- Separation logic is well-matched with graph acyclicity
- Cannot allow Iris invariants: need to control resource transfer

Lessons learnt for deadlock-free Iris

- Benchmark: deadlock freedom for session types using *semantic typing* with Actris
- In general, *memory leak freedom* and *deadlock freedom* are equally hard (thanks to POPL reviewers)
- Graph acyclicity is key for higher-order deadlock freedom
- Separation logic is well-matched with graph acyclicity
- Cannot allow Iris invariants: need to control resource transfer

Ideas? Questions?

`julesjacobs@gmail.com`

Extra slides

Stronger deadlock and leak freedom result

Global progress is the standard notion that people use

Our POPL reviewers: Can your method prove something stronger?

Stronger deadlock and leak freedom result

Global progress is the standard notion that people use
Our POPL reviewers: Can your method prove something stronger?

Partial deadlock: a set S of threads and channels such that:

1. All threads in S are blocked on a channel in S
2. No references to channels in S from outside S

Stronger deadlock and leak freedom result

Global progress is the standard notion that people use
Our POPL reviewers: Can your method prove something stronger?

Partial deadlock: a set S of threads and channels such that:

1. All threads in S are blocked on a channel in S
2. No references to channels in S from outside S

Strong reachability:

1. A channel is reachable if it is referenced by a reachable channel or thread
2. A thread is reachable if it can step, or is blocked on a reachable channel

Stronger deadlock and leak freedom result

Global progress is the standard notion that people use
Our POPL reviewers: Can your method prove something stronger?

Partial deadlock: a set S of threads and channels such that:

1. All threads in S are blocked on a channel in S
2. No references to channels in S from outside S

Strong reachability:

1. A channel is reachable if it is referenced by a reachable channel or thread
2. A thread is reachable if it can step, or is blocked on a reachable channel

Lemma. All threads and channels are reachable $\iff$ no partial deadlock

Stronger deadlock and leak freedom result

Global progress is the standard notion that people use
Our POPL reviewers: Can your method prove something stronger?

Partial deadlock: a set S of threads and channels such that:

1. All threads in S are blocked on a channel in S
2. No references to channels in S from outside S

Strong reachability:

1. A channel is reachable if it is referenced by a reachable channel or thread
2. A thread is reachable if it can step, or is blocked on a reachable channel

Lemma. All threads and channels are reachable $\iff$ no partial deadlock
Lemma. Any thread or channel is reachable $\implies$ global progress

Stronger deadlock and leak freedom result

Global progress is the standard notion that people use
Our POPL reviewers: Can your method prove something stronger?

Partial deadlock: a set S of threads and channels such that:

1. All threads in S are blocked on a channel in S
2. No references to channels in S from outside S

Strong reachability:

1. A channel is reachable if it is referenced by a reachable channel or thread
2. A thread is reachable if it can step, or is blocked on a reachable channel

Lemma. All threads and channels are reachable $\iff$ no partial deadlock
Lemma. Any thread or channel is reachable $\implies$ global progress
Theorem. For well-typed initial programs, no partial deadlock occurs

L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, 2010. doi: 10.1007/978-3-642-15375-4_16. URL https://doi.org/10.1007/978-3-642-15375-4_16.

D. Castro-Perez, F. Ferreira, and N. Yoshida. EMTST: engineering the meta-theory of session types. In *TACAS (2)*, volume 12079 of *LNCS*, pages 278–285, 2020. doi: 10.1007/978-3-030-45237-7_17. URL https://doi.org/10.1007/978-3-030-45237-7_17.

L. Ciccone and L. Padovani. A dependently typed linear π-calculus in agda. In *PPDP*, pages 8:1–8:14, 2020. doi: 10.1145/3414080.3414109. URL https://doi.org/10.1145/3414080.3414109.

S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):19–50, 2010. doi: 10.1017/S0956796809990268. URL https://doi.org/10.1017/S0956796809990268.

M. A. Goto, R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely. An extensible approach to session polymorphism. *MSCS*, 26(3):465–509, 2016. doi: 10.1017/S0960129514000231. URL https://doi.org/10.1017/S0960129514000231.

J. K. Hinrichsen, J. Bengtson, and R. Krebbers. Actris: Session-type based reasoning in separation logic. *PACMPL*, 4(POPL), Dec. 2020. doi: 10.1145/3371074. URL https://doi.org/10.1145/3371074.

J. K. Hinrichsen, D. Louwrink, R. Krebbers, and J. Bengtson. Machine-checked semantic session typing. In *CPP*, pages 178–198, 2021. doi: 10.1145/3437992.3439914. URL https://doi.org/10.1145/3437992.3439914.

K. Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523, 1993. doi: 10.1007/3-540-57208-2_35. URL https://doi.org/10.1007/3-540-57208-2_35.

R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217, 2017. doi: 10.1145/3009837.3009855. URL https://doi.org/10.1145/3009837.3009855.

S. Lindley and J. G. Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *LNCS*, pages 560–584, 2015. doi: 10.1007/978-3-662-46669-8_23. URL https://doi.org/10.1007/978-3-662-46669-8_23.

A. Rouvoet, C. Bach Poulsen, R. Krebbers, and E. Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP*, 2020. ISBN

9781450370974. doi: 10.1145/3372885.3373818. URL
https://doi.org/10.1145/3372885.3373818.

P. Thiemann. Intrinsically-typed mechanized semantics for session types. In *PPDP*, 2019. ISBN 9781450372497. doi: 10.1145/3354166.3354184. URL https://doi.org/10.1145/3354166.3354184.

P. Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012. doi: 10.1145/2364527.2364568. URL https://doi.org/10.1145/2364527.2364568.