Deadlock Freedom for Session Types Using Separation Logic

Jules Jacobs

Radboud University Nijmegen
julesjacobs@gmail.com

Joint work with Robbert Krebbers (RU) and Stephanie Balzer (CMU)

Overview

- Channels for concurrency
- Operational semantics
- Session types
- Goal of the research project
- Typing channel references with separation logic
- The connectivity graph
- Deadlock freedom

Channels for concurrency

Example:

```
let c' = fork(fun c =>
    send(c, 3)
    let k = recv(c)
    if k < 10
      then send(c,"hello")
      else send(c,"hi")
    close(c)
  )
let n = recv(c')
send(c', 2*n)
let msg = recv(c')
close(c')
print(msg)
```

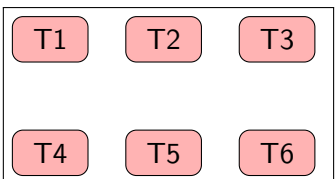
Channel operations:

- `let c' = fork(fun c => ...)`
- `send(c, msg)`
- `let msg = recv(c)`
- `close(c)`

Operational model

State of the system: thread pool + channel buffers

Threads:



Channel buffers:

channel 0:	$msg_{01}, msg_{02}, \ldots$	$msg'_{01}, msg'_{02}, \ldots$
channel 1:	$msg_{11}, msg_{12}, \ldots$	$msg'_{11}, msg'_{12}, \ldots$
channel 2:	$msg_{21}, msg_{22}, \ldots$	$msg'_{21}, msg'_{22}, \ldots$

There are two buffers per channel!

In the program `let c' = fork(fun c => ...)`,
one buffer is for the messages from `c` to `c'`,
and the other is for messages from `c'` to `c`.

Potential bugs

- Wrong type of message, e.g. string instead of int **(type error)**:

```
let c' = fork(fun c => send(c,"hi"); ...)
recv(c') + 3
```

- Using a channel after close **(use-after-free)**:

```
close(c); let msg = recv(c)
```

- Both sides do a receive **(deadlock)**:

```
let c' = fork(fun c => recv(c); ...)
recv(c')
```

TODO: More examples of deadlock

- Not closing a channel **(leak)**:

```
fork(fun c => recv(c))
```

Ruling out these bugs is tricky:

Higher order channels: `let c = fork(...) in send(d,c)`

Closures that capture channels: `let f = (fun x => ... c ...) in send(d,f)`

Session types (Honda et al. 1998)

Session types (Honda et al. 1998) are a type system for ruling out these bugs.
A **linear** type system: **variables must be used exactly once.**

Examples:

```
let x = ... in
... x ...
```

$\Longrightarrow$ **OK!**

```
let x = ... in
... x ... x ...
```

$\Longrightarrow$ **TYPE ERROR!**

```
let x = ... in
... (no x) ...
```

$\Longrightarrow$ **TYPE ERROR!**

Send and receive must use functional style:

```
let c2 = send(c1,msg)
let (c3,msg) = recv(c2)
close(c3)
```

Close is **required** to get rid of variable c3!

Rules out 2 of 4 potential bugs immediately:

- Not closing a channel (leak)
  Because variable c3 **must** be used.
- Using a channel after close (use-after-free)
  Because the use `close(c3)` **consumes** variable c3.

Session types: **channel protocols**

Example: `!Int, ?Bool, !String, End`

Means: first send an `Int`, then receive a `Bool`, then send a `String`, then close the chanel.

```
send : (!A,R) x A -> R        fork[R] : (dual(R) -> Unit) -> R
recv : (?A,R) -> R x A        close : End -> Unit
```

```
let c1 = fork[!Int, ?Bool, !String, End](fun c1' => ...)
  // c1  : !Int, ?Bool, !String, End
  // c1' : ?Int, !Bool, ?String, End
let c2 = send(c1,3)
  // c2 : ?Bool, !String, End
let (c3,b) = recv(c2)
  // c3 : !String, End
let c4 = send(c3, "hello")
  // c4 : End
close(c4)
```

Rules out local deadlocks (both sides try to receive)

Features & Extensions

Closures & higher order channels:

```
?(!String, ?Int, End), !(Bool -> Int), End
```

Recursive channel protocols:

```
T = ?Int, !String, T
```

Choice in channel protocols:

```
T = (?Int, !String, End) + (!Bool, ?Int, End)
```

Dependent channel protocols:

```
T = ?(i:Int) if i < 10 then (!String, End) else (?Bool, End)
```

Asynchronous subtyping:

```
(?Bool, !Int, R) <: (!Int, ?Bool, R)
```

Shared channels: controlled non-linear usage of channels protected by locks.

```
let l = newlock(c)
let d = fork(fun d' => ... l ...) in ... l ...
```

Research goal (ongoing research)

Session types rule out:

- **Type errors**
- **Use-after-free**
- **Leaks**
- **Deadlocks? Session types do rule out all deadlocks!** (Honda et al. 1998)

Our goal: a proof of these 4 properties,

- Mechanised proof in Coq
- For a low-level operational semantics with buffers
- Reasoning about channel ownership at a high level using separation logic
- Supporting both locks and channels using **connectivity graphs**

Rest of the talk:

(1) What the formal statement of the theorems is like

(2) High level overview of the proof: separation logic & connectivity graph

Operational semantics (1)

- **State of the system:** $\text{state} = \overbrace{\text{list expr}}^{\text{threads}} \times (\overbrace{\mathbb{N} \times \text{Bool} \rightarrow_{\text{fin}} \text{list val}}^{\text{buffers}})$
- **Grammar of expressions:**

$$
\begin{aligned}
\text{expr} ::= \ & \text{val} \\
| \ & \text{expr} + \text{expr} \\
| \ & \text{let x} = \text{expr in expr} \\
| \ & \textbf{send(expr, expr)} \\
| \ & \textbf{recv(expr, expr)} \\
| \ & \textbf{fork(expr)} \\
| \ & \textbf{close(expr)} \\
| \ & \ldots
\end{aligned}
$$

- **Values:** $\text{val} ::= \text{n:nat} \mid (\text{val}, \text{val}) \mid \text{fun x} \implies \text{expr} \mid \textbf{chan(n:nat,b:bool)} \mid \ldots$

Operational semantics (2)

Small-step operational semantics with inductively defined relations:

- $\text{pure\_step} \subseteq \text{expr} \times \text{expr}$
- $\text{head\_step} \subseteq (\text{expr} \times \text{buffers}) \times (\text{expr} \times \text{buffers} \times \overbrace{\text{list expr}}^{\text{spawned threads}})$
- $\text{ctx} \subseteq \text{expr} \to \text{expr}$
- $\text{ctx\_step} \subseteq (\text{expr} \times \text{buffers}) \times (\text{expr} \times \text{buffers} \times \text{list expr})$
- $\text{step} \subseteq (\text{list expr} \times \text{buffers}) \times (\text{list expr} \times \text{buffers})$

Operational semantics (3)

In Coq:

```
Inductive head_step :
    expr -> heap -> expr -> heap -> list expr -> Prop :=
| Pure_step : forall e e' h,
  pure_step e e' -> head_step e h e' h []
| Send_step : forall h c y buf,
   h !! (other c) = Some buf ->
   head_step (Send (Val (ChanV c)) (Val y)) h
             (Val (ChanV c)) (<[ other c := buf ++ [y] ]> h) []
| ...
```

Type system rules (1)

- **Grammar of types:**

$$\begin{aligned} \text{type} ::= & \ \text{unit} \\ & \mid \text{type} \times \text{type} \\ & \mid \text{type} \to \text{type} \\ & \mid \text{chan\_type} \\ & \mid \ldots \end{aligned}$$

- **Channel protocols:**

$$\begin{aligned} \text{chan\_type} ::= & \ !\text{type}, \text{chan\_type} \\ & \mid ?\text{type}, \text{chan\_type} \\ & \mid \text{End} \\ & \mid \ldots \end{aligned}$$

Type system rules (2)

The type system is also inductively defined:

- Judgement: $\Gamma \vdash e : t$
- $(\cdot \vdash \cdot : \cdot) \subseteq \text{env} \times \text{expr} \times \text{type}$
- $\text{env} = \text{var} \rightarrow_{\text{fin}} \text{type}$

$$\frac{\cdot}{\{x \mapsto t\} \vdash x : t}\ \textsc{Var} \qquad \frac{n \in \mathbb{N}}{\emptyset \vdash n : \mathit{Nat}}\ \textsc{Nat}$$

$$\frac{\Gamma_1 \vdash e_1 : t_1 \to t_2 \qquad \Gamma_2 \vdash e_2 : t_1 \qquad \Gamma_1 \cap \Gamma_2 = \emptyset}{(\Gamma_1 \cup \Gamma_2) \vdash e_1\ e_2 : t_2}\ \textsc{App}$$

$$\frac{\Gamma_1 \vdash e_1 : !t, R \qquad \Gamma_2 \vdash e_2 : t \qquad \Gamma_1 \cap \Gamma_2 = \emptyset}{(\Gamma_1 \cup \Gamma_2) \vdash \mathsf{send}(e_1, e_2) : R}\ \textsc{Send}$$

Properties that we want to prove

- Suppose we have a top level well-typed program $\emptyset \vdash e : \text{unit}$
- We initialize the state with $S_0 = ([e], []) \subseteq \text{threads} \times \text{buffers}$
- We then run the operational semantics step $\subseteq (\text{threads} \times \text{buffers}) \times (\text{threads} \times \text{buffers})$ to get a new state $S' = ([e_1, e_2, \ldots], \{(0, \mathit{true}) \mapsto \mathit{buf}_{00}, (0, \mathit{false}) \mapsto \mathit{buf}_{01}, \ldots\})$

We want to establish these properties:

- Each expression e_i in S' either
  1. Is a unit value, or
  2. Is waiting on recv(chan(n,b), v), or
  3. Can take a step
- If S' can not take *any* further step, then
  1. All the e_i are unit values, and
  2. All the buffers are empty

These properties rule out type errors, use-after-close, deadlocks, and leaks.

Method: progress & preservation

Idea of progress & preservation:

Choose a set of well-behaved states $W \subseteq (\text{threads} \times \text{buffers})$.

- Show that $S_0 \in W$
- **Preservation:** Show that if $S \in W$ and $(S, S') \in \text{step}$, then $S' \in W$.
- **Progress:** Show that the properties that we want hold for states in W.

What do we want of W?

Choose W such that $([e_1, e_2, \dots], \{(0, \mathit{true}) \mapsto \mathit{buf}_{00}, (0, \mathit{false}) \mapsto \mathit{buf}_{01}, \dots\}) \in W$ if

- All the expressions e_i are well-typed.
- All the values in the buffers are well-typed.
- For each channel $n \in \mathbb{N}$, there is exactly one $\text{chan}(n, \mathit{true})$ and one $\text{chan}(n, \mathit{false})$.
  - These channel references may occur inside one of the e_i or inside one of the buffers.
  - The types of the $\mathit{chan}(n, b)$ must be consistent with the values inside the buffers.
- The connectivity between channels and threads is *acyclic*.
  - Otherwise we can have a deadlock due to cyclic waiting.

Two type systems: user facing type system & run-time type system

- The initial expression e does not have any $chan(n, b)$ in it
- Once the operational semantics steps, these channel references apprear inside the e_i
- And inside the buffers (when sending channels over channels)

Problem: The type system does not know what to do with $chan(n, b)$
Solution: Define an extended type system that can handle $chan(n, b)$

Extended type system judgement: $\Gamma, \Sigma \vdash e : t$

- Γ tells us the types of variables
- Σ tells us the types of $chan(n, b)$'s

Both Γ, Σ split up *disjointly* over subexpressions.

Run-time type system: separation logic

Inference rule of extended type system:

$$\frac{\Gamma_1, \Sigma_1 \vdash e_1 : t_1 \to t_2 \qquad \Gamma_2, \Sigma_2 \vdash e_2 : t_1 \qquad \Gamma_1 \cap \Gamma_2 = \emptyset \qquad \Sigma_1 \cap \Sigma_2 = \emptyset}{(\Gamma_1 \cup \Gamma_2), (\Sigma_1 \cup \Sigma_2) \vdash e_1\ e_2 : t_2}$$

We use **separation logic** to hide the Σ's:

$$\frac{\Gamma_1 \vdash e_1 : t_1 \to t_2 \qquad * \qquad \Gamma_2 \vdash e_2 : t_1 \qquad \Gamma_1 \cap \Gamma_2 = \emptyset}{(\Gamma_1 \cup \Gamma_2) \vdash e_1\ e_2 : t_2}\ *$$

Instead of an **implication** with **conjunction** between the premises,
we have a **magic wand** with **separating conjection** between the premises.

We need to transfer ownership of channels as the expressions take steps, and as values get put in and taken out of buffers.

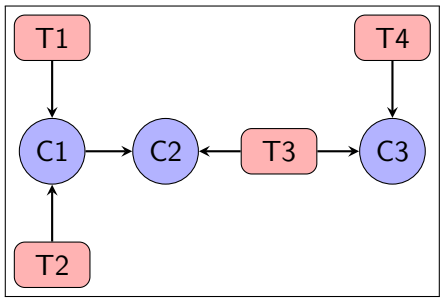
Separation logic allows us to reason about linear channel ownership at a high level.

Low-level view: Σ is split up and re-distributed over parts of the state as the state evolves.

High-level view: separation logic invariant hold, ownership gets transfered.

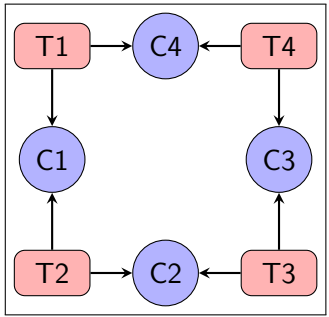
The connectivity graph

- A graph containing threads (boxes) and channels (circles)
- An arrow $T \rightarrow C$ means that thread T has a reference to channel C
  - This means that T contains a channel reference to C
- An arrow $C1 \rightarrow C2$ means that channel $C1$ has a reference to channel $C2$
  - This means that somewhere in the buffer of $C1$ there is a reference to $C2$



Deadlocks

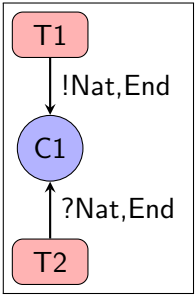
Cyclic connectivity graphs can result in deadlock:



T1 waits to receive from C1, then sends to C4
T2 waits to receive from C2, then sends to C1
T3 waits to receive from C3, then sends to C2
T4 waits to receive from C4, then sends to C3

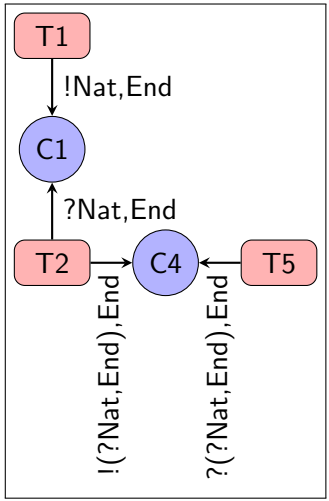
Invariant: the connectivity graph is acyclic

The evolution of the connectivity graph: initial



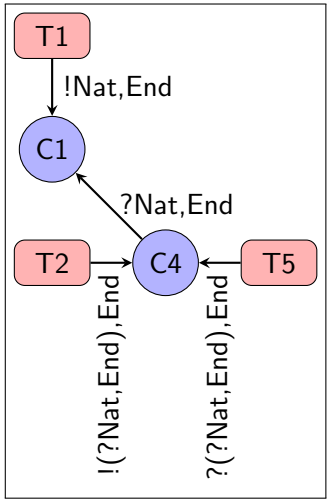

```
own_edge T1 c1 (true,t)
own_edge T2 c1 (false,dual(t))

c1 : !(?Nat, End), End.
T1 := let c1' = send(c1',3); close(c1')
T2 :=
  let c4 = fork(fun c4' =>
    let (c4',c1) = recv(c4')
    let (c1,n) = recv(c1)
```

The evolution of the connectivity graph: initial



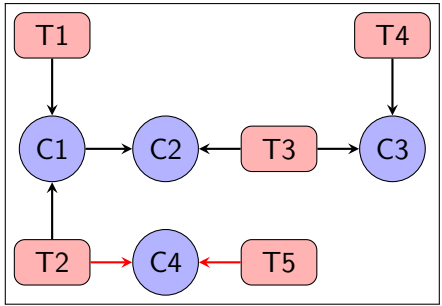

```
own_edge T1 c1 (true,(!Nat,End))
own_edge T2 c1 (false,(?Nat,End)
own_edge T2 c4 (true,(!(!Nat,End),End))
```

The evolution of the connectivity graph: initial

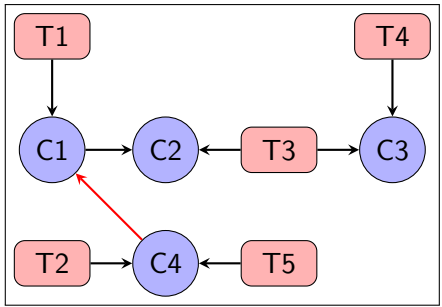



```
own (c1,true) T1 (!Nat,End)
own (c1,false) c4 (?Nat,End)
own (c4,true) T2 (!(!Nat,End),End)
```

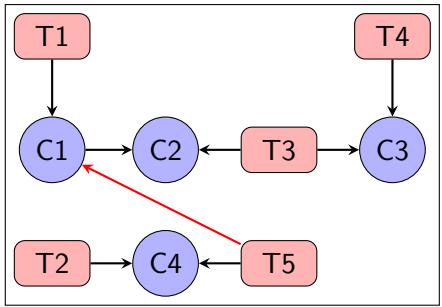
The evolution of the connectivity graph: fork



The evolution of the connectivity graph: send C1 over C4

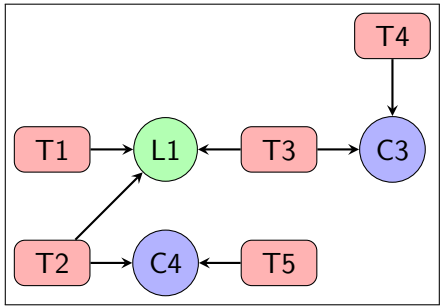


The evolution of the connectivity graph: receive C1 over C4



Concurrent objects

We can generalize from channels to other concurrent objects, such as locks.



Required property: no local deadlock

- Channels: local deadlock means both sides do a receive
  - Ruled out by type system
- Locks: local deadlock means all threads are waiting on the lock
  - Can't happen: one of the threads must have succesfully acquired the lock

Connectivity graph is a tree: no local deadlock $\implies$ no deadlock

Coq proof

The Coq proof is work in progress...

- **Language & type system**
- **Separation logic**
- **Run-time type system**
- **Type-safety invariant**
- **Library for tree reasoning**
- **Preservation of type safety invariant**
- **Preservation of tree invariant**
- **Invariant $\Longrightarrow$ desired properties**
- **Extensions**

Questions?