Mechanized deadlock freedom for session types

Jules Jacobs, Stephanie Balzer, Robbert Krebbers

May 6, 2021

Abstract

We prove deadlock freedom for a language with channels and session types. Our operational semantics is realistic, and simulates message sends with explicit buffers and threads. Our proof maintains a logical *connectivity graph* in parallel with the physical program state, with the invariant that the connectivity graph is acyclic and the program state is well-typed. Our proof is structured to be amenable to mechanization in the Coq proof assistant.

1 Introduction

Session types [Hon93, HVK98] form a system for message-passing concurrency in which bidirectional channels follow protocols that are chosen when a channel is created. Session types ensure type safety even for programs that use channels in a higher-order fashion (*i.e.,* that send channel endpoints and closures over channels). Additionally, if channel creation is done simultaneously with thread creation so that the current thread and the newly spawned thread have a communication channel between them, then session types also guarantee deadlock and leak freedom. That is, they guarantee that it is impossible to end up in a situation where all threads are waiting for another thread, or where channels are not freed when all threads of the program have terminated.

While there are many results on deadlock and leak freedom for session types [CP10, Wad12, TCP13, LM15, CP20]—including extensions to challenging type systems with concepts like sharing [BTP19]—these proofs are notoriously subtle and have never been mechanized in a proof assistant. Proofs typically involve reasoning to show that the dependency structure of the threads and channels is acyclic (which is key to establish deadlock and leak freedom). This reasoning is intertwined with reasoning about the programming language's operational semantics and session-type system.

The goal of our work is to decouple these two concepts to obtain a modular and extensible proof, which is moreover mechanizable in a proof assistant. To decouple these two concepts, we introduce the notion of a *connectivity graph*, which provides an abstract account of the dependency structure of channels and threads. The vertices of a connectivity graph generalize threads and channels, while the edges generalize the concept of channel ownership. Deadlock freedom and leak freedom are captured abstractly by acyclicity of the connectivity graph. Using a proof in the style of progress and preservation [WF94, Pie02, Har16] we connect our abstract connectivity graphs to the concrete operational semantics and type system. We prove preservation of the message-passing operations using canonical transformations on connectivity graphs. We prove progress using a generic theorem about connectivity graphs to find a thread that can take a step.

Using our new notion of connectivity graphs, we show that in addition to giving a more modular proof of deadlock and leak freedom, we can go beyond the state of the art in a number of ways. First, we consider a more realistic language with an operational semantics that is closer to the way message passing is commonly implemented—we consider an ML-style language with asynchronous semantics, a low-level operational semantics (using a flat thread pool and a flat heap of buffers instead of structural congruences), and other programming language features (like lambda abstractions and general recursion). Second, we mechanize all our results in the Coq proof assistant (existing mechanizations of type safety of session types [Thi19, RBPKV20, HLKB21] do not establish deadlock and leak freedom).

2 Setting: a concurrent linear λ-calculus with session types

Before demonstrating our abstract notion of a connectivity graph, this section briefly describes the concrete programming language that we study. We consider a λ-calculus with multiple threads and communication via bidirectional channels. Its abstract syntax is:

$$\begin{aligned} e \in \mathsf{Expr} ::= {} & x \mid n \mid e\ e \mid \lambda x.\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \cdots \mid \\ & \mathsf{fork}(e) \mid \mathsf{send}(e, e) \mid \mathsf{receive}(e) \mid \mathsf{close}(e) \end{aligned}$$

We use an asynchronous semantics, with two buffers per channel to guarantee that sends in either direction are non-blocking. The semantics of fork, send, receive, and close is as expected:

$\mathsf{fork}(\lambda x.\ e)$ Allocates a new channel with two endpoints $c_{\mathsf{left}}, c_{\mathsf{right}}$, and starts a new thread running $[c_{\mathsf{left}}/x]e$, where the endpoint c_{left} is substituted for x in e. The fork expression itself returns the endpoint c_{right}.

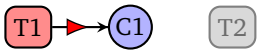
$\mathsf{send}(c, v)$ Places the message v into the appropriate buffer of endpoint c and returns c.[1]

$\mathsf{receive}(c)$ Takes a message v out of the appropriate buffer of endpoint c and returns the pair (c, v). If the buffer is empty, it blocks until a message is available.

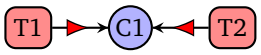
$\mathsf{close}(c)$ Closes the endpoint c and returns unit.

Untyped programs in our language can deadlock. A set of threads is in a *deadlock* if each member of the set is permanently blocked from making progress. On the left we have three programs, and on the right we have a graphical depiction of the resulting deadlocked state:

```
// no counter party
let c1left = fork(λ c1right, ())
receive(c1left)
```



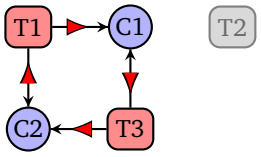

```
// protocol violation
let c1left = fork(λ c1right, receive(c1right); ...)
receive(c1left)
```




```
// circular dependency
let c1left = fork(λ c1right, send(c1right,c1right))
let (c1left,c1right) = receive(c1left)

let c2left = fork(λ c2right,
  let (c2right,v) = receive(c2right)
  send(c1right,v); ...)

let (c1left,v) = receive(c1left)
  send(c2left,v)
```



Whereas a program only mentions channel endpoints, a state figure accounts for the channels themselves (depicted as a circles) and depicts references to channel endpoints by *black arrows*. To reason about deadlocks, we annotate reference arrows with *red triangles*. A triangle points to the side that is responsible for carrying out the next action. For example, a thread with a red triangle pointing to a channel is currently waiting to receive a message from that channel.

As demonstrated by the above programs, a deadlock does not necessarily have to be a circular dependency (third program). If threads violate the usual protocol that one side receives and the other side sends a message, then a deadlock can occur if both try to receive (second program). The simplest form of deadlock is a thread attempting to receive a message from a channel that nobody else has a reference to (first program).

[1] The reason why send returns the endpoint c is the session type system, which gives the endpoint a new type, prescribing the remainder of the protocol. The same applies to receive.

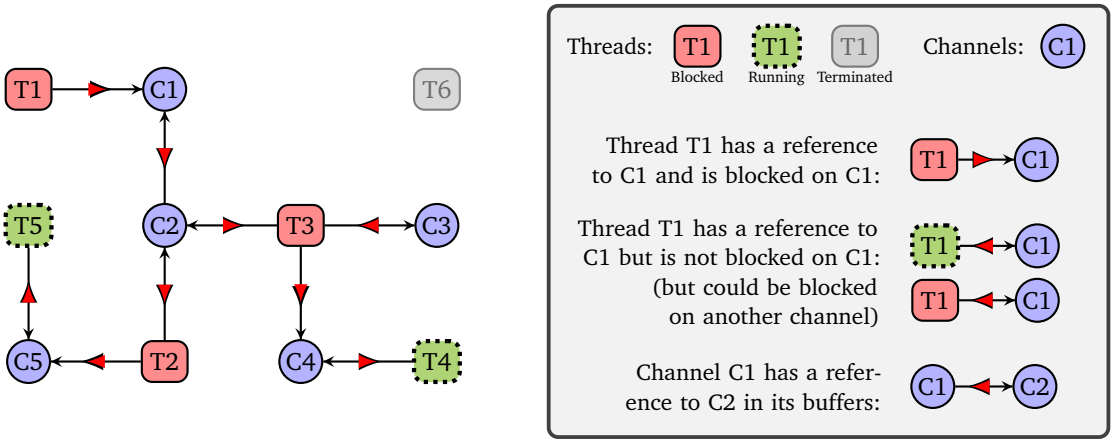


Figure 1: Connectivity graph: example (left) and description (right).

A linear type system with session types can be used to rule out deadlocks:

$$t \in \mathsf{Type} ::= \mathbf{1} \mid \mathbf{N} \mid t \times t \mid t \to t \mid s$$

$$s \in \mathsf{Session} ::= ?t.s \mid !t.s \mid \mathsf{End}$$

A session type denotes a sequence of actions, with $?t$ indicating a receive, $!t$ a send, and End termination, where t denotes the type of the message. The language is *higher-order* because it allows sending any value over a channel, including closures and channel endpoints.

A key ingredient of deadlock freedom is that channel creation and thread spawning are combined such that the two endpoints are given to two concurrently running threads. This is enforced by our fork construct. If this property is complemented with linear typing of channels, the resulting reference structure of a well-typed program state is *acyclic*, ruling out deadlocks by construction. Concretely, the deadlocks shown above are ruled out by several properties of the linear session-type system:

1. The first one is ruled out by ensuring that there always exists a counter party (no weakening)
2. The second one is ruled out by ensuring that all threads adhere to protocols (session duality)
3. The third one is ruled out by ensuring that the reference structure is acyclic (no contraction)

To modularly reason about deadlock freedom, we abstract the reference structure of a program state by a *connectivity graph* and prove deadlock freedom in terms of acyclicity of this connectivity graph. We introduce the connectivity graph in the next section.

3 The connectivity graph

In the context of a low level language semantics like ours, formally defining what it means to have a deadlock and proving the absence of deadlocks is subtle. The key contribution of our work is the notion of a *connectivity graph*, which can be used to give an abstract account of deadlock freedom and is a crucial part of our proof.

The vertices of the connectivity graph are threads and channels, and the edges depict the *reference structure*. We have an edge $T \longrightarrow C$ from a thread to a channel if the thread T has ownership of (*i.e.,* holds a reference to) the channel C. We have an edge $C \longrightarrow C'$ from a channel to another channel if a reference to C' occurs in one of the message buffers of C.

To account for the waiting dependencies between threads and channels, we label the edges of the connectivity graph with *red triangles*, indicating the *waiting direction*, as follows:

- If T is blocked on C, then the waiting direction is the same as the reference direction.
- For all other edges, the waiting direction is the opposite of the reference direction.

This defines the connectivity graph of a given program state. Figure 1 gives a graphical example and a description of the types of arrows that can occur in the connectivity graph.

We can deduce from the connectivity graph whether a deadlock has occurred. In this case, it has not, because every thread that is blocked is transitively waiting on threads that are still running. Thread T1 is blocked on receiving a message from C1. The channel C1 is in turn stored in a buffer of C2, so before C1 can provide a message to T1, it first needs to escape from C2's buffer. The channel C2 is waiting for either thread T2 or T3 to perform a receive, and those two threads are themselves blocked on C5 and C4, but the other endpoints of those channels belong to threads T5 and T4 that are still running. Channel C3 has only one endpoint left, and is waiting for thread T3 to take any remaining messages out of its buffer, and gets deallocated when its last endpoint is closed.

We define *deadlock freedom* to mean that the waiting graph (red triangles) is a directed acyclic graph (DAG) and its leaves are threads that can take a step.

Deadlock freedom implies:

1. *Global progress:* there exists a thread that can take a step, or all threads are terminated.
2. *Local progress:* all threads can make a step, are blocked on a receive, or are terminated.
3. *Leak freedom:* there are no channels that are not connected to any thread.

The structure of the connectivity graph changes as new channels and threads get created, messages containing channel references get put into buffers and taken out on the other side, and channels get deallocated. Each of these operations induces a local transformation of the connectivity graph, and we must ensure that these rewrites maintain acyclicity. To accommodate such transformations, our formalization equips the connectivity graph with canonical graph transformations that can be used to model the corresponding transitions in our operational semantics. The next section illuminates how we link the connectivity graph to the concrete program state and phrase *invariants* to be maintained along transitions to keep them in sync.

4 The invariant preserved by the operational semantics

We state the requirement that the connectivity graph remains acyclic along program transitions as a *global invariant*. This invariant is endangered by channel operations (*i.e.,* send, receive, fork, and close) and thus is connected to the session types of the channel endpoints. The types of the two endpoints of a channel must be dual to each other, modulo the messages queued up in the buffers, a property that we impose as a *local invariant*. To express the local invariant we use the typing judgment $\Gamma; \Sigma \vdash e : t$, which allows us to type the state of a program being executed. As is usual, the linear context Γ provides the typing of variables, and the linear context Σ provides the session types of channel references. The local invariants then are:

- For a thread with expression e, the local invariant is $\emptyset; \Sigma \vdash e : ()$.
- For a channel with buffers $(\vec{v_L}, \vec{v_R})$ and channel types (σ_L, σ_R) on the incoming edges the local invariant is $\Sigma \vdash_{\mathsf{buf}} (\vec{v_L}, \vec{v_R}) : (\sigma_L, \sigma_R)$. The judgment $(\vdash_{\mathsf{buf}})$ is defined in terms of $(\vdash)$ and expresses that the session types σ_L, σ_R are dual up to the values $(\vec{v_L}, \vec{v_R})$ in the buffers, which themselves contain the channel references in Σ.

To tie the local invariants to the global invariant, we label each edge of the connectivity graph with its session type. The set of outgoing labeled edges of a vertex forms its local Σ-context. The proof of deadlock freedom then proceeds in the style of progress and preservation [WF94, Pie02, Har16]:

1. We prove that the global invariant is satisfied for a configuration consisting of a single initial thread with a well-typed expression $\emptyset \vdash e : ()$.
2. We prove *preservation*: the global invariant is preserved when the operational semantics takes a step. Preservation of the message-passing operations (*i.e.,* fork, send, receive, and close) is proved using theorems that establish that the generic graph transformations preserve acyclicity. The different cases are depicted in Figure 2.
3. We prove *progress*: if a program state satisfies the global invariant, then deadlock freedom holds for that program state. They key ingredient of progress is phrased in terms of a generic theorem about the connectivity graph to find a thread that can take a step.

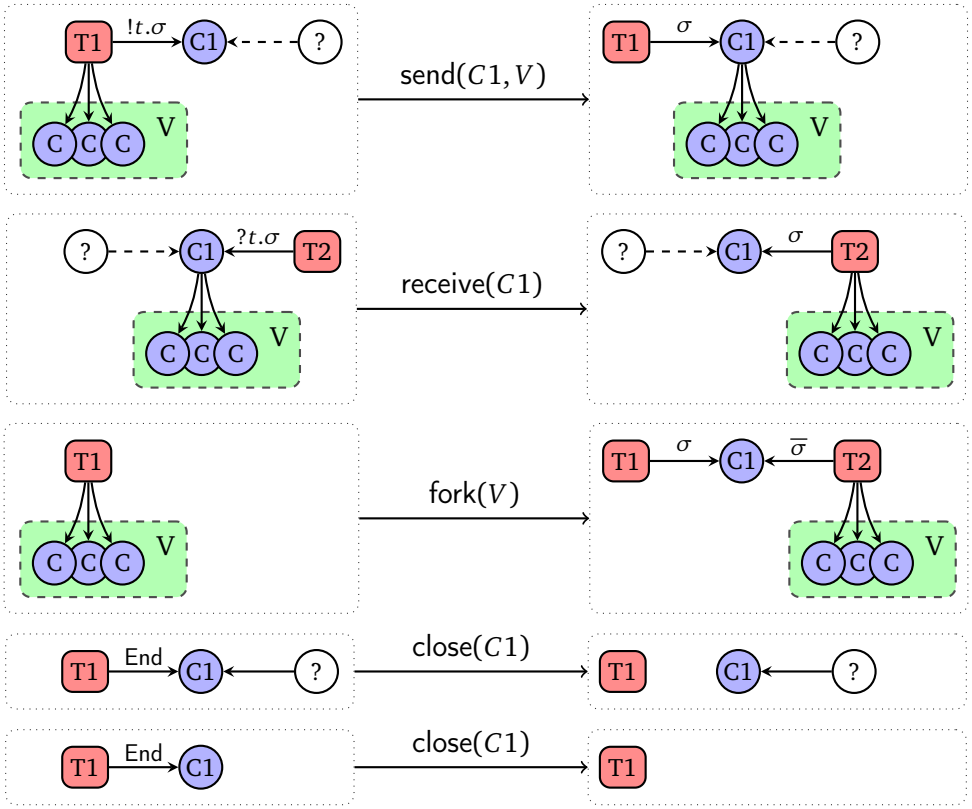


Figure 2: The operational steps and their effect on the connectivity graph.

5 Conclusion

We have sketched a proof of deadlock freedom based on the notion of a *connectivity graph*. Our goal is to mechanize this proof in Coq. To boost compositionality, we abstract our connectivity graph even further into a *undirected acyclic graph* that is agnostic of the kinds of vertices (*i.e.,* thread or channel), and then describe transformations on the connectivity graph in terms of graph transformations. Moreover, we equip this graph with a generic reachability argument that finds a leaf (*i.e.,* a thread that can take a step). So far we have mechanized this graph with supporting lemmas to reason about acyclicity and reachability as a library in Coq. Moreover, we have mechanized the definition of the language and type system, type preservation and thread local progress. It remains to connect this library to the operational semantics via the connectivity graph.

References

[BTP19] Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *ESOP*, 2019. doi:10.1007/978-3-030-17184-1_22.

[CP10] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, 2010. doi:10.1007/978-3-642-15375-4_16.

[CP20] Luca Ciccone and Luca Padovani. A dependently typed linear π-calculus in Agda. In *PPDP*, 2020. doi:10.1145/3414080.3414109.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, USA, 2nd edition, 2016. doi:10.5555/3002812.

[HLKB21] Jonas Kastberg Hinrichsen, Daniël Louwrink, Robbert Krebbers, and Jesper Bengtson. Machine-checked semantic session typing. In *CPP*, 2021. doi:10.1145/3437992.3439914.

[Hon93] Kohei Honda. Types for dyadic interaction. In *CONCUR*, 1993. doi:10.1007/3-540-57208-2_35.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, 1998. doi:10.1007/BFb0053567.

[LM15] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, 2015. doi:10.1007/978-3-662-46669-8_23.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. doi:10.5555/509043.

[RBPKV20] Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Intrinsically-typed definitional interpreters for linear, session-typed languages. In *CPP*, 2020. doi:10.1145/3372885.3373818.

[TCP13] Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, 2013. doi:10.1007/978-3-642-37036-6_20.

[Thi19] Peter Thiemann. Intrinsically-typed mechanized semantics for session types. In *PPDP*, 2019. doi:10.1145/3354166.3354184.

[Wad12] Philip Wadler. Propositions as sessions. In *ICFP*, 2012. doi:10.1145/2364527.2364568.

[WF94] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *I&C*, 115(1), 1994. URL: https://www.sciencedirect.com/science/article/pii/S0890540184710935, doi:https://doi.org/10.1006/inco.1994.1093.