GUARANTEES BY CONSTRUCTION

Types for deadlock and leak free concurrency • separation logics for verified message passing • general and efficient coalgebraic automata minimization • paradox-free probabilistic programming.

JULES JACOBS

ABSTRACT

This thesis is about type systems for deadlock and leak free concurrency, separation logics for verified message passing, paradox-free probabilistic programming, and general and efficient coalgebraic automata minimization. In each case we aspire to guarantee beneficial properties 'by construction', as inherent consequences of the design of the system:

TYPES FOR DEADLOCK AND MEMORY LEAK-FREE CONCURRENCY:
Languages such as Rust guarantee memory safety and race freedom by type checking, but Rust programs can deadlock and leak memory. By enforing a carefully designed linear typing discipline, we obtain new concurrent languages with locks and channels in which deadlock freedom and memory leak freedom are guaranteed by type checking. A key challenge is proving this formally, for which we introduce a proof technique based on *connectivity graphs*.

SEPARATION LOGICS FOR VERIFIED MESSAGE PASSING:
As preparation, we show that concurrent separation logic and message passing are a perfect match, by using nested invariants to markedly simplify the soundness proof of an Actris-style separation logic for the verification of message passing programs. We then develop a new separation logic for verifying that message passing programs are deadlock free and memory leak free. These properties follow automatically from the linearity of the separation logic, without any additional proof obligations.

GENERAL AND EFFICIENT COALGEBRAIC AUTOMATA MINIMIZATION:
A variety of automaton types and minimization algorithms exist, often tailored to specific cases. By imposing a coalgebraic structure on our automata, we can exploit their shared characteristics. This allows us to develop a unified minimization algorithm that works across a general class of automata, while being efficient both in theory and in practice.

PARADOX-FREE PROBABILISTIC PROGRAMMING:
Probabilistic modeling languages suffer from paradoxes when conditioning on events of measure zero. This can even lead to different answers depending on whether the modeler is working in metric or imperial units. We show that we can avoid these paradoxes with a simple change to the modeling language. The resulting language guarantees that all probabilistic programs are invariant under change of parameterization, and its semantics of conditioning on events of measure zero relate to conditioning on events of positive measure in a natural way.

Contents

Part I

INTRODUCTION

Part II

TYPES FOR DEADLOCK AND LEAK FREE CONCURRENCY

Chapter 1

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

ABSTRACT We introduce the notion of a *connectivity graph*—an abstract representation of the topology of concurrently interacting entities, which allows us to encapsulate generic principles of reasoning about *deadlock freedom*. Connectivity graphs are *parametric* in their vertices (representing entities like threads and channels) and their edges (representing references between entities) with labels (representing interaction protocols). We prove deadlock and memory leak freedom in the style of progress and preservation and use *separation logic* as a meta theoretic tool to treat connectivity graph edges and labels substructurally. To prove preservation locally, we distill generic separation logic rules for *local graph transformations* that preserve acyclicity of the connectivity graph. To prove global progress locally, we introduce a *waiting induction* principle for acyclic connectivity graphs. We mechanize our results in Coq, and instantiate our method with a higher-order binary session-typed language to obtain the first mechanized proof of deadlock and leak freedom.

1.1 INTRODUCTION

Binary session types (Honda, 1993; Honda et al., 1998) are a type discipline for specifying protocols of interactions in message-passing concurrent programs. Session types have turned into an active area of research that enjoys strong theoretical and practical foundations. The theoretical foundations include a Curry-Howard correspondence between session-typed π-calculi and linear logic (Caires and Pfenning, 2010; Wadler, 2012; Caires et al., 2013; Pérez et al., 2014; Toninho et al., 2013; Lindley and Morris, 2015; Toninho, 2015) and session-typed λ-calculi with mainstream programming language features (Lindley and Morris, 2016b, 2017; Igarashi et al., 2017; Fowler et al., 2019). The practical foundations include libraries for session types in mainstream programming languages (Dezani-Ciancaglini et al., 2006; Pucella and Tov, 2008; Imai et al., 2010; Jespersen et al., 2015; Lindley and Morris, 2016a; Scalas and Yoshida, 2016; Padovani, 2017; Imai et al., 2019; Kokke, 2019; Chen and Balzer, 2020).

Session-typed languages come with strong guarantees: they not only enjoy *type safety* (a.k.a. session fidelity) but all well-typed programs also enjoy *deadlock freedom* (and consequently, *global progress*). The proofs of deadlock freedom have to establish that the dependency structure among the threads (or processes) and channels (or buffers) remains *acyclic*, even in the presence of dynamic thread spawning and

higher-order channels (Carbone and Debois, 2010). Despite the active developments in the mechanization of the meta-theory of binary session types (Thiemann, 2019; Rouvoet et al., 2020; Hinrichsen et al., 2021b; Tassarotti et al., 2017; Goto et al., 2016; Ciccone and Padovani, 2020; Castro-Perez et al., 2020; Gay and Vasconcelos, 2010), a mechanized proof of deadlock freedom for binary session types with dynamic thread and channel creation and a dynamically changing communication topology (due to higher-order channels) is still outstanding because of the intricacies of reasoning about graphs in a mechanized setting. While the semantics of global and local types of multiparty session types has recently been mechanized (Castro-Perez et al., 2021), and thus global properties such as deadlock freedom shown to hold, the result is confined to a single session without dynamic thread and channel creation and without higher-order channels.

In this paper we develop a parametric proof method for deadlock freedom of concurrently computing entities that interact via shared resources on a dynamically changing acyclic communication topology. We mechanize the proof method in the Coq proof assistant, and instantiate it for a deadlock freedom proof for a variant of GV (Wadler, 2012; Lindley and Morris, 2015), a functional language with higher-order binary linear session types. Proof mechanization has the obvious benefit of providing the peace of mind of a machine-checked proof. Another—maybe even more important—benefit of mechanization is that it encourages us to develop abstractions that encapsulate the reasoning about the acyclic dependency structure of threads and channels, and that shield us from the intricacies of a language's operational semantics and type system.

The key ingredients that make our proof method parametric are our new notion of a *connectivity graph*, to abstract over the dependency structure, and our meta theoretic use of *separation logic* (O'Hearn et al., 2001), to link our abstract connectivity graph to the concrete language's operational semantics and type system. A connectivity graph abstracts concurrent entities and shared resources as vertices and their possible interactions as edges, which are labeled with protocol state. When instantiating the connectivity graph for session types, threads and channels become vertices, channel references become edges whose labels indicate the session type of the referenced channel. By asserting *acyclicity* of the connectivity graph, circular dependencies among the concurrent entities and shared resources are rendered impossible. This guarantees that at any moment at least one interaction can happen (deadlock freedom) and that all channels are deallocated when the program terminates (memory leak freedom).

EXAMPLE Before we explain the parametric aspects of our proof method, let us consider an example program to see connectivity graphs for linear session types in action:

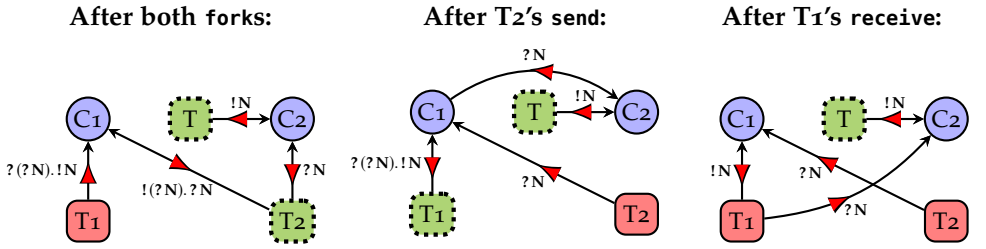


Figure 1: Connectivity graphs with run-time information for our example program (the End markers have been elided from session types). Boxes depict threads (red boxes are blocked threads, green dotted boxes are running threads). Blue cicles depict channels. Black edges indicate references to channel endpoints, labeled with their session type. Red triangles reveal the waiting dependency for each reference to a channel endpoint: either the owner of the endpoint is waiting to receive a message from the channel, or the channel is waiting for the owner of the endpoint to initiate the next action (send or receive or close).

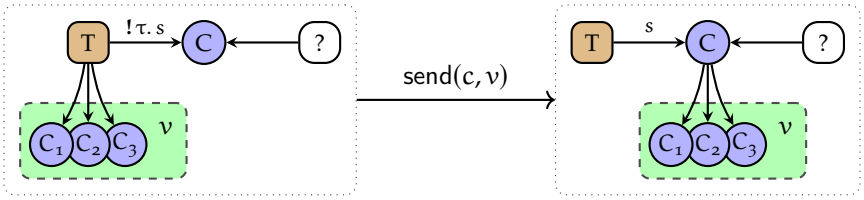
```
1  let c1 = fork (λ c1',                          // c1': ?(?N.End).!N.End
2    let (c1',c) = receive(c1')                   // c1': !N.End, c: ?N.End
3    let (c,n) = receive(c); ...)                 // c1': !N.End, c: End
4                                        T1
5  let c2 = fork (λ c2',                          // c1: !(?N.End).?N.End, c2': ?N.End
6    let c1 = send(c1,c2');                       // c1: ?N.End
7    let (c1,m) = receive(c1); ...)               // c1: End
8                                          T2
9                                                 // c2: !N.End
10 let c2 = send(c2,10); ...                      // c2: End
                                           T
```

The main thread (T) uses the `fork` construct to spawn two threads (T1 and T2) with bidirectional channels (C1 and C2) connecting them to the main thread. The endpoints `c2` and `c2'` of channel C2 (created on Line 5) have session types (**!N**.End) and (**?N**.End), respectively. These dual session types express that a number should be sent (!) over `c2` and received (?) over `c2'`. The session types of channel C1 (created on Line 1) are more interesting—they are higher-order. Endpoint `c1` has session type (**!**(**?N**.End).**?N**.End), which expresses that first a channel of type (**?N**.End) should be sent, after which a number can be received. The λ-expression of thread T2 captures endpoint `c1`, resulting in the ownership of `c1` being transferred from thread T to thread T2.

The first picture in Figure 1 displays the connectivity graph after both `fork`s have been executed, but no other steps have been performed yet. The solid red boxes correspond to threads that are blocked on a receive, while the dotted green boxes correspond to threads that can make progress. The small black arrowheads on the edges indicate the direction of channel ownership: an edge from a thread to a channel indicates that the thread owns an endpoint of that channel, and

an edge from a channel to a channel indicates that an endpoint of the latter channel is stored in one of the buffers of the former channel (an edge between two threads is not possible—all references are *to* channels). The red triangles denote the waiting dependency. A crucial property of the connectivity graph is that the waitchapters/cgraphs/paper/transformations.texg dependency remains acyclic. Acyclicity enables us to find a thread that can make progress by starting at any vertex and repeatedly following the red triangles. For example, when starting at thread T1, which is blocked, we find that thread T2 can perform a step.

When we continue by letting thread T2 perform the `send` operation on Line 6, the `send` will move the endpoint `c2'` into the buffer of C1. In general, the effect of the `send` operation on the connectivity graph is as follows:



On the left, thread T has ownership of the transmitted value v and the endpoint of carrier channel C with session type $!\tau.\, s$. The value v could in general contain any number of channel references (depicted as C_1, C_2, C_3), for instance when it is a pair of channels or a closure that has captured more than one channel. On the right, we have the resulting connectivity graph that we obtain after the `send` operation has been performed. The session type changes to s (*i.e.*, the remainder of the protocol) and the value v gets transferred to the buffer of C, so the incoming edges of the channels in v are changed from T to C. Note that the red waiting triangles and the information about whether a thread is blocked is not part of the connectivity graph because it can be derived from the run-time configuration. We therefore depict the general transformation rule without waiting triangles and use a neutral color for threads. In our running example, the thread is T2, the channel is C1, and the value v is the single channel C2. The second picture in Figure 1 displays the resulting connectivity graph: the session type of T2 has advanced to $(?\,\mathbf{N}.\,\mathrm{End})$ and the incoming edge from C2 to T2 has turned into an incoming edge from C2 to C1.

Next, we let thread T1 perform the `recv` operation on Line 2, which will move the endpoint out of the buffer of channel C1 and bind it to variable `c`. The rule to transform the connectivity graph for a `receive` operation is similar to `send` (the exact rule can be found in Figure 8 in Section 1.3.5). The third picture in Figure 1 displays the resulting connectivity graph, where we see that the session type of T1 advanced, and that the outgoing edge from C1 to C2 has turned into an outgoing edge from T1 to C2. Observe that the connectivity graph has a non-trivial structure—to find a thread that can unblock T2, we need to follow multiple edges to end up in thread T.

PROGRESS AND PRESERVATION As shown by the examples in Figure 1, connectivity graphs describe the types and abstract reference topology of a program's execution configuration, but not the concrete expressions and values that constitute the threads and channels. To prove a property about the operational semantics, we need to define a relation that expresses that a configuration ρ is well-formed w.r.t. a connectivity graph G. With that relation at hand, we can carry out a proof in the style of progress and preservation (Wright and Felleisen, 1994; Harper, 2016; Pierce, 2002).

- **Progress**: If ρ is well-formed w.r.t. G, then either ρ is final (all threads have terminated and all channels have been deallocated), or ρ can step (deadlock freedom).
- **Preservation**: If ρ is well-formed w.r.t. G, and can take a step $\rho \rightsquigarrow \rho'$ in the operational semantics, then we can transform G into G' so that ρ' is well-formed w.r.t. G'.

It is important to point out that connectivity graphs generalize *heap typings* from traditional progress and preservation proofs for type systems with mutable references (Pierce, 2002; Harper, 2016). Whereas heap typings are flat (they merely give the types of channels, which suffices for type safety), connectivity graphs additionally describe the reference topology and ensure its acyclicity (needed for deadlock and memory leak freedom).

CONNECTIVITY GRAPHS AS A PARAMETRIC PROOF PRINCIPLE When trying to formalize the above reasoning, we encounter two problems:

1. Due to linear types and concurrency, it is non-trivial to formalize the well-formedness relation of configurations w.r.t. connectivity graphs. Definitions easily end up cluttered with details about *resource separation*, which burdens mechanization in a proof assistant.
2. Proving preservation and progress involves non-trivial reasoning about graphs. For preservation we need to *transform* graphs (to type a post-configuration), and for progress we need to *traverse* graphs (to find a thread that can step). Reasoning about graphs is difficult in a proof assistant because graphs are not inductively defined.

To address these problems, we use separation logic (O'Hearn et al., 2001) as a meta theoretic tool to reason about graphs. Traditionally, separation logic is used as a specification language to write pre- and postconditions for individual programs in Hoare-style logics. Inspired by recent work that uses separation logic to establish type safety using logical relations (Krebbers et al., 2017; Jung et al., 2018a; Hinrichsen et al., 2021b) and intrinsically-typed interpreters and compilers (Rouvoet et al., 2020, 2021), we also use separation logic but in the context of a progress and preservation proof.

Our version of separation logic makes it possible to define the well-formedness relation in a way that is local (*i.e.,* talks about threads in isolation) and that hides resources. To adopt separation logic for our connectivity graph, we must decide on what to consider as a resource. The scenarios in Figure 1 suggest that we should consider a vertex's *outgoing edges* as resources, because then a graph transformation, such as the one induced by moving endpoint `c2'` into C1's buffer, simply amounts to an ownership transfer. To prove preservation, we distill a set of separation logic rules for reasoning about *graph transformations* by simply transferring ownership of resources. To prove progress, we distill a form of *waiting induction* to perform induction on the connectivity graph to find a vertex that can perform a step.

All ingredients of our method (the definition of connectivity graph, the separation logic, the graph transformations, and waiting induction) are parametric in the vertices, edges and labels of the connectivity graph. This is crucial for mechanization: we can encapsulate our proof method as a library that is independent of the concrete programming language. We use our library in combination with the Iris Proof Mode (Krebbers et al., 2017, 2018)—which provides tactics for separation-logic based reasoning—to effectively hide reasoning about graphs and resources in Coq.

CONTRIBUTIONS We present a parametric method for proving deadlock and memory leak freedom of binary linear session-typed languages. Concretely:

- We introduce *connectivity graphs* as a generalization of heap typings in progress and preservation proofs. In addition to typing, connectivity graphs track the reference topology.
- We show how to use separation logic in a non-standard way as a language for linking our abstract connectivity graphs to a concrete language's operational semantics and type system.
- We implement connectivity graphs as a library in the Coq proof assistant that is parametric in the type of vertices and edges. Our library includes *graph transformations* as separation logic rules to aid proving preservation, and a principle of *waiting induction* over connectivity graphs to aid proving progress.
- We use our connectivity graph library to obtain the first mechanized proof of deadlock and leak freedom for a binary session-typed λ-calculus with higher-order channels, recursive types, and unrestricted types.

We start by introducing our language (Section 1.2), and explain our key ideas by proving deadlock freedom for it (Section 1.3). Next, we present the parametric aspects of our proof method (Sections 1.4 and 1.5). We then add extensions to our language, and prove a stronger deadlock and memory leak freedom property than the conventional formulation in terms of global progress (Section 1.6), and describe our Coq mechanization (Section 1.7). We finish with related and future work (Sections 1.8 and 1.9). An archive of the Coq mechanization can be found at Jacobs et al. (2021), and the most recent version at `https://github.com/julesjacobs/cgraphs`.

1.2 LANGUAGE AND OPERATIONAL SEMANTICS

We present the core of our session-typed λ-calculus with concurrency and asynchronous bidirectional channels (extensions with more features are described in Section 1.6). This language is inspired by GV (Wadler, 2012; Lindley and Morris, 2015), but there are a couple of differences. First, we are more liberal and allow both channel endpoints to be closed anytime, rather than only when a thread terminates. For our proofs this extension poses no problem—it just means that our connectivity graphs might become disconnected. Second, our operational semantics uses a flat thread pool and heap rather than binders and structural congruences, resembling more closely a realistic implementation of message passing. The syntax of expressions of our core language is:

$$\begin{aligned} e \in \mathsf{Expr} ::= {} & x \mid () \mid n \mid (e, e) \mid \lambda x.\ e \mid c \mid e\ e \mid \mathsf{let}\ () = e\ \mathsf{in}\ e \mid \mathsf{let}\ (x_1, x_2) = e\ \mathsf{in}\ e \mid \\ & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{fork}(e) \mid \mathsf{send}(e, e) \mid \mathsf{receive}(e) \mid \mathsf{close}(e) \mid \cdots \end{aligned}$$

The literals include the unit value (), numbers $n \in \mathbb{N}$, and channel endpoint references $c \in \mathsf{Chan}$ (these enter expressions at run time, see the operational semantics below). As usual in a linearly-typed language, we consider let-binding constructs $\mathsf{let}\ () = e\ \mathsf{in}\ e$ and $\mathsf{let}\ (x_1, x_2) = e\ \mathsf{in}\ e$ for pattern matching on the unit value () and pairs (e, e), respectively.

OPERATIONAL SEMANTICS. We use an asynchronous semantics with two buffers per channel to guarantee that sends in either direction are non-blocking.[1] This is formally modeled as:

$$\begin{aligned} c \in \mathsf{Chan} ::= {} & \#(a, t) & \qquad h \in \mathsf{Heap} \triangleq \mathsf{Chan} \xrightarrow{\mathsf{fin}} \mathsf{List}\,\mathsf{Val} \\ v \in \mathsf{Val} ::= {} & () \mid n \mid (v, v) \mid \lambda x.\ e \mid c & \qquad \rho \in \mathsf{Cfg} \triangleq \mathsf{List}\,\mathsf{Expr} \times \mathsf{Heap} \end{aligned}$$

A heap h is a finite map from channel endpoint references to buffers (modeled as lists of values). Channel endpoint references $\#(a, t)$ consist of an address $a \in \mathsf{Addr}$ and a tag $t \in \{0, 1\}$ denoting the endpoint. The operation $\overline{\#(n, t)} \triangleq \#(n, 1 - t)$ gives the opposite endpoint. Configurations $(\vec{e}, h)$ consist of a list of expressions $\vec{e}$, modeling the threads, and a heap h that is shared by these threads. The semantics of most constructs is standard, so we focus on the message passing constructs:

$\mathsf{fork}(v)$ Allocates a new channel with endpoints $c_{\mathsf{left}} \triangleq \#(a, 0)$ and $c_{\mathsf{right}} \triangleq \#(a, 1)$, where a is a fresh address. It starts a new thread running $v\ c_{\mathsf{left}}$ (v should be a function) and returns c_{right}.

1 Due to the session typing discipline, only one of the buffers is expected to be populated at any given time. The two buffers are important to distinguish the origin of the messages, because otherwise an asynchronous send followed by a receive creates a risk that the thread receives back its own message that it just sent.

Pure reduction relation:

$$
\begin{aligned}
(\lambda x.\ e)\ v &\rightsquigarrow_{\text{pure}} e[v/x] \\
\mathsf{let}\ x = v\ \mathsf{in}\ e &\rightsquigarrow_{\text{pure}} e[v/x] \\
\mathsf{let}\ () = ()\ \mathsf{in}\ e &\rightsquigarrow_{\text{pure}} e \\
\mathsf{let}\ (x_1, x_2) = (v_1, v_2)\ \mathsf{in}\ e &\rightsquigarrow_{\text{pure}} e[v_1/x_1][v_2/x_2] \\
\mathsf{if}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &\rightsquigarrow_{\text{pure}} e_1 && (\text{if } n \neq 0) \\
\mathsf{if}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 &\rightsquigarrow_{\text{pure}} e_2 && (\text{if } n = 0)
\end{aligned}
$$

Head reduction relation:

$$
\begin{aligned}
(e_1, h) &\rightsquigarrow_{\text{head}} (e_2, h, \epsilon) && (\text{if } e_1 \rightsquigarrow_{\text{pure}} e_2) \\
(\mathsf{fork}(v), h) &\rightsquigarrow_{\text{head}} (\#(a,1), h \uplus \{(a,0) \mapsto \epsilon, (a,1) \mapsto \epsilon\}, [v\ \#(a,0)]) \\
& && (\text{if } (a,0), (a,1) \notin \mathsf{dom}(h)) \\
(\mathsf{send}(c, v), h \uplus \{\overline{c} \mapsto \vec{v}\}) &\rightsquigarrow_{\text{head}} (c, h \uplus \{\overline{c} \mapsto \vec{v} +\!\!+ [v]\}, \epsilon) \\
(\mathsf{receive}(c), h \uplus \{c \mapsto [v] +\!\!+ \vec{v}\}) &\rightsquigarrow_{\text{head}} ((c, v), h \uplus \{c \mapsto \vec{v}\}, \epsilon) \\
(\mathsf{close}(c), h \uplus \{c \mapsto \epsilon\}) &\rightsquigarrow_{\text{head}} ((), h, \epsilon)
\end{aligned}
$$

Global reduction relation:

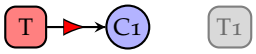
$$
\begin{aligned}
(\vec{e_a} +\!\!+ [\mathsf{K}[\, e\,]] +\!\!+ \vec{e_b}, h) &\rightsquigarrow_{\text{global}} (\vec{e_a} +\!\!+ [\mathsf{K}[\, e'\,]] +\!\!+ \vec{e_b} +\!\!+ \vec{e}, h') \\
& && (\text{if } (e, h) \rightsquigarrow_{\text{head}} (e', h', \vec{e}))
\end{aligned}
$$

Evaluation contexts:

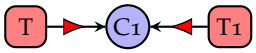
$$
\begin{aligned}
\mathsf{K} \in \mathsf{Ctx} ::= {} & \square \mid (\mathsf{K}, e) \mid (v, \mathsf{K}) \mid \mathsf{K}\ e \mid v\ \mathsf{K} \mid \mathsf{let}\ x = \mathsf{K}\ \mathsf{in}\ e \mid \mathsf{let}\ () = \mathsf{K}\ \mathsf{in}\ e \mid \mathsf{let}\ (x_1, x_2) = \mathsf{K}\ \mathsf{in}\ e \mid \\
& \mathsf{if}\ \mathsf{K}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{fork}(\mathsf{K}) \mid \mathsf{send}(\mathsf{K}, e) \mid \mathsf{send}(v, \mathsf{K}) \mid \mathsf{receive}(\mathsf{K}) \mid \mathsf{close}(\mathsf{K})
\end{aligned}
$$

Figure 2: The operational semantics of our language.

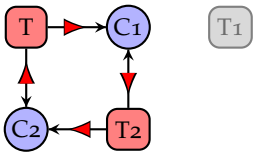
```
// no counter party
let c1 = fork(λ c1', ())
receive(c1)
```




```
// protocol violation
let c1 = fork(λ c1', receive(c1'); ...)
receive(c1)
```




```
// circular dependency
let c1 = fork(λ c1', send(c1', c1'))
let (c1,c1') = receive(c1)
let c2 = fork(λ c2',
    receive(c2'); send(c1', 2); ...)
receive(c1); send(c2, 3)
```




```
// memory leak
let c1 = fork(λ c1', ())
let c2 = fork(λ c2', ())
send(c2, c1)
send(c1, c2)
```

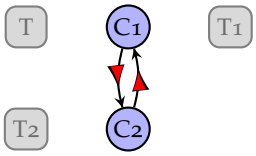


Figure 3: Examples of configurations that are deadlocked or have leaked.

`send(c, v)` Places the message v into the buffer of the opposite endpoint $\overline{c}$ of c and returns c.[2] This construct does not block.

`receive(c)` Takes a message v out of the buffer of endpoint c and returns the pair (c, v). If the buffer is empty, it blocks until a message is available.

`close(c)` Closes the endpoint c and returns the unit value $()$. This construct does not block.

The formal definition of the semantics is given in Figure 2. It involves three reduction relations: (1) pure reductions $e \rightsquigarrow_{\text{pure}} e'$, (2) head-reductions $(e, h) \rightsquigarrow_{\text{head}} (e', h', \vec{e})$, where $\vec{e}$ is a list of spawned threads (empty for all constructs but fork), and (3) global reductions $\rho \rightsquigarrow_{\text{global}} \rho'$. Global reductions make use of standard call-by-value evaluation contexts $K \in \text{Ctx}$.

DEADLOCKS AND MEMORY LEAKS. Untyped programs in our language can deadlock or have memory leaks. A configuration $(\vec{e}, h)$ is *deadlocked* if each expression $e \in \vec{e}$ is a receive(c) that is waiting on an empty buffer c in the heap h. A configuration $(\vec{e}, h)$ has *leaked* if each expression $e \in \vec{e}$ is a value, but the heap h is not empty, meaning not all channels have been closed. In Figure 3 we show examples of both. On the left we show the code, and on the right we show a graphical representation of

2 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 the receive operation.

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

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

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \uplus \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{1} \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ () = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2 \uplus \{x_1 \mapsto \tau_1\} \uplus \{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ (x_1, x_2) = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\frac{\Gamma_1 \vdash e_1 : \mathbf{N} \qquad \Gamma_2 \vdash e_2 : \tau \qquad \Gamma_2 \vdash e_3 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau} \qquad \frac{\Gamma \vdash e : \overline{s} \multimap \mathbf{1}}{\Gamma \vdash \mathsf{fork}(e) : s}$$

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

Figure 4: The static type system of our language.

the resulting configuration. As in Section 1.1, boxes denote threads (*i.e.*, expressions), circles denote channels (*i.e.*, buffer pairs), black arrows denote channel references, and red triangles denote the waiting dependency. Concretely, a thread with a red triangle pointing to a channel is waiting to receive a message from that channel, and a channel with a red triangle pointing to a thread is waiting for the thread to send a message along that channel.

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). 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). A deadlock can occur even if all parties are locally well behaved, but cause a cyclic dependency (third program). Note that even though the reference structure (black arrows) of this example forms a directed acyclic graph, a deadlock occurs because the waiting direction (red triangles) can be opposite of the reference direction (black arrows). Finally, memory leaks can occur if channels are not properly closed (fourth program).

SESSION TYPING. A linear type system with session types can be used to rule out deadlocks:[3]

$$\tau \in \mathsf{Type} ::= \mathbf{1} \mid \mathbf{N} \mid \tau \times \tau \mid \tau \multimap \tau \mid s$$
$$s \in \mathsf{Session} ::= \mathsf{End} \mid \mathbf{?}\,\tau.\,s \mid \mathbf{!}\,\tau.\,s$$

A session type s denotes a sequence of actions, with $\mathbf{?}\,\tau$ indicating a receive, $\mathbf{!}\,\tau$ a send, and End termination, where τ denotes the type of the message. The *dual* $\overline{s}$ of a session type s is defined by flipping all sends (**!**) and receives (**?**):

$$\overline{\mathsf{End}} \triangleq \mathsf{End} \qquad \overline{\mathbf{!}\,\tau.\,s} \triangleq \mathbf{?}\,\tau.\,\overline{s} \qquad \overline{\mathbf{?}\,\tau.\,s} \triangleq \mathbf{!}\,\tau.\,\overline{s}$$

The rules of the type system are shown in Figure 4. Note that the type system is *higher-order* because it allows sending any value over a channel, including functions and channel endpoints.

Session types ensure deadlock and leak freedom by combining channel and thread creation through the fork construct.[4] Together with linear channel typing, this ensures that the resulting reference structure is *acyclic*, even when viewed as an *undirected graph*, in which edges may be traversed in either direction. Let us consider the deadlocked programs in Figure 3. The first one is ruled out by ensuring that there always exists a counter party (due to the absence of weakening). The second one is ruled out by ensuring that all threads adhere to protocols (due to session duality). The third one is ruled out by ensuring that the reference structure is acyclic (due to the absence of contraction). The memory leak in the last example is ruled out by a combination of these rules.

1.3 KEY IDEAS

Before we detail the abstractions that make our proof method parametric (Section 1.4 and Section 1.5), we describe a concrete instantiation of our method to our session-typed language (Section 1.2). To do so, we first discuss the well-known method of progress and preservation and the challenges in applying it to prove deadlock and resource leak freedom (Section 1.3.1). To address these challenges, we introduce connectivity graphs (Section 1.3.2) and describe how we use separation logic to formalize run-time typing judgments for individual expressions (Section 1.3.3) and a well-formedness predicate for configurations (Section 1.3.4). We finally show how to use our proof method to prove preservation (Section 1.3.5) and progress (Section 1.3.6).

3 For simplicity we let even integers be linear; in Section 1.6 we extend the type system with support for unrestricted types.

4 If we had a $\mathsf{new\_chan} : \mathbf{1} \multimap (s \times \overline{s})$ construct, then $\mathsf{let}\ (c_1, c_2) = \mathsf{new\_chan}()\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{receive}(c_1)\ \mathsf{in}\ \ldots$ would deadlock.

1.3.1 *Generalizing The Progress and Preservation Method*

Traditionally, a language is said to be *type safe* if well typed programs do not get stuck. For purely functional languages, like the Simply Typed Lambda Calculus (STLC), this is stated as:

Theorem 1.3.1 (Type safety)**.** *If* $\emptyset \vdash e : \tau$ *and* $e \rightsquigarrow^* e'$*, then either* e' *is a value, or* e' *can step further (i.e.,* $\exists e''.\, e' \rightsquigarrow e''$*).*

Type safety is often proved using the method of *progress and preservation* (Wright and Felleisen, 1994; Harper, 2016; Pierce, 2002), which decomposes type safety into two properties that imply it:

- **Preservation:** If $\emptyset \vdash e : \tau$ and $e \rightsquigarrow e'$, then $\emptyset \vdash e' : \tau$.
- **Progress:** If $\emptyset \vdash e : \tau$, then either e is a value, or e can step (*i.e.,* $\exists e''.\, e' \rightsquigarrow e''$).

For pure languages like STLC, the proofs of these properties are straightforward: both properties are proved by induction on the structure of the typing judgment and/or the reduction relation.

For languages with mutable state or concurrency, the above properties must be generalized to account for a program's run-time configurations. In general, for a language with expressions Expr we have a set $\rho \in \mathsf{Cfg}$ of configurations, an initial configuration $\mathsf{init} \in \mathsf{Expr} \to \mathsf{Cfg}$, and a predicate $\mathsf{final} \in \mathsf{Cfg} \to \mathsf{Prop}$ of configurations that are considered to be safely terminated.

Theorem 1.3.2 (Generalized type safety)**.** *If* $\emptyset \vdash e : ()$ *and* $\mathsf{init}(e) \rightsquigarrow^* \rho$*, then either* $\mathsf{final}(\rho)$ *or* ρ *can step further (i.e.,* $\exists \rho'.\, \rho \rightsquigarrow \rho'$*).*

Recall that for our session-typed language we have $\mathsf{Cfg} \triangleq \mathsf{List}\ \mathsf{Expr} \times \mathsf{Heap}$. We let final select configurations where all threads have terminated with a unit value, and the heap is empty:

$$\mathsf{init}(e) \triangleq ([e], \emptyset) \qquad\qquad \mathsf{final}(\vec{e}, h) \triangleq h = \emptyset \wedge \forall i.\, e_i = ()$$

By defining final this way, the type safety theorem expresses *deadlock and memory leak freedom*.[5] To see why, consider a configuration $\rho = (\vec{e}, h)$ that does not satisfy $\mathsf{final}(\rho)$ and cannot step any further. It must either consist of threads $\vec{e}$ that have not terminated but cannot step (indicating a deadlock), or of terminated threads $\vec{e}$ but a non-empty heap h (indicating a memory leak).

For deadlock and resource leak freedom, we need to restrict expressions to have a ground type. For example, an expression like $\mathsf{fork}(\lambda x.\, \mathsf{close}(x)) : \mathsf{End}$ exhibits a trivial memory leak because the channel endpoint returned by $\mathsf{fork}()$ is still active.

5 This kind of deadlock freedom is also known as *global progress* in the session type literature. In Section 1.6.3 we prove a stronger property that also rules out partial deadlocks.

For simplicity, we use the unit type () in Theorem 1.3.2, but of course, other ground types like **N** would suffice too.

The method of progress and preservation can be generalized to prove our generalized type safety theorem (Theorem 1.3.2) by choosing a well-formedness predicate $\mathsf{wf} \in \mathsf{Cfg} \to \mathsf{Prop}$ that satisfies:

- **Initialization:** If $\emptyset \vdash e : ()$, then $\mathsf{wf}(\mathsf{init}(e))$.
- **Generalized preservation:** If $\mathsf{wf}(\rho)$ and $\rho \rightsquigarrow \rho'$, then $\mathsf{wf}(\rho')$.
- **Generalized progress:** If $\mathsf{wf}(\rho)$, then either $\mathsf{final}(\rho)$ or ρ can step further (*i.e.*, $\exists \rho'.\, \rho \rightsquigarrow \rho'$).

The primary challenge is to define a well-formedness predicate wf in such a way that these properties can be proved. A naive definition of $\mathsf{wf}(\vec{e}, h)$ would simply demand each expression in the thread pool $\vec{e}$ and each buffer in the heap h to be well typed. Unfortunately, this naive definition does not quite work:

1. Channel references $\#(a, t)$ enter the expressions $\vec{e}$ and heap h throughout the execution of the program. The typing judgment $\Gamma \vdash e : \tau$ of our type system (Figure 4) does not account for channel references $\#(a, t)$ because they cannot be written in source programs.
2. Whenever a channel reference $\#(a, t)$ appears in a thread or channel buffer, the type of $\#(a, t)$ should match up with the values in the buffers at address a in the heap h, and with the type $\#(a, 1 - t)$ of the other endpoint.

For the simpler case of proving type safety for the STLC with references, Harper (2016) and Pierce (2002) remedy issue (1) by introducing a *run-time typing judgment* $\Gamma; \Sigma \vdash e : \tau$. This judgment extends the static typing judgment with a *heap typing* Σ, which assigns types to heap addresses. Issue (2) is addressed because the heap typing makes sure that the typing of each reference is consistent with the corresponding value in the heap.

Unfortunately, adapting this approach to prove deadlock and resource freedom is not as simple. Conventional heap typings only capture the typing of addresses, not the acyclicity of the reference topology. The latter is crucial to prove "generalized progress", which states that the well-formedness predicate wf indeed implies deadlock and resource leak freedom.

1.3.2 *Generalizing Heap Typings to Connectivity Graphs*

Our notion of *connectivity graphs* generalizes the notion of heap typings by simultaneously keeping track of the types of channels in the heap, and providing an abstract representation of the reference topology. In their full generality, connectivity graphs

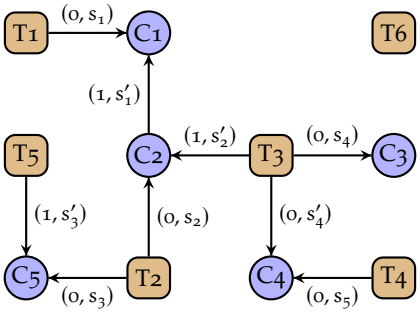


Figure 5: An example of a connectivity graph. Brown boxes depict threads, and blue circles depict channels.

are represented as finite maps from pairs of vertices V to the labels L on the edges between them:

$$G \in \mathsf{Cgraph}(V, L) \triangleq \left\{ G \in V \times V \xrightarrow{\text{fin}} L \;\middle|\; G \text{ has no undirected cycles} \right\}$$

To reason about our language defined in Section 1.2, we instantiate the vertices V and edge labels L of a connectivity graph $\mathsf{Cgraph}(V, L)$ as follows:

$$v \in V ::= \mathsf{Thread}(i) \mid \mathsf{Chan}(\alpha) \qquad\qquad l \in L \triangleq \{0, 1\} \times \mathsf{Session}$$

The vertices V are threads $\mathsf{Thread}(i)$ (with position i in the thread pool) or channels $\mathsf{Chan}(\alpha)$ (with address α in the heap). The edges are channel references and have a label $(t, s) \in L$ that consists of a tag $t \in \{0, 1\}$, indicating the channel endpoint pointed to, and a session type $s \in \mathsf{Session}$, indicating the endpoint's session type. An example of a connectivity graph is depicted in Figure 5. Note that the red triangles that we used to depict the waiting directions in Figures 1 and 3 in Section 1.1 are not part of the connectivity graph itself, because these can be derived from the run-time configuration. The session types on the edges *are* part of the connectivity graph, because they cannot be derived from the run-time configuration. In Theorem 1.3.4 we formalize how the red triangles are derived.

Connectivity graphs corresponding to configurations in our language have a number of important properties. First, vertices $\mathsf{Thread}(i)$ can have an arbitrary number of outgoing edges, but have no incoming edges. That is because threads can own channel endpoints, but threads can never be owned. Second, vertices $\mathsf{Chan}(\alpha)$ can also have an arbitrary number of outgoing edges, but at most two incoming edges. Outgoing edges are due to higher-order channels—a channel c_1 can be sent over another channel c_2, resulting in an edge from c_2 to c_1 that models that c_2 owns c_1. Ingoing edges correspond to a channel's endpoints, which can be at most two. The number of incoming edges is one in case one channel endpoint has been deallocated, but the other is still in use.

A key ingredient of connectivity graphs is the acyclicity restriction. They should be *acyclic* in the *undirected* sense: there must be no cycles even if we disregard the direction of the edges. In other words, a connectivity graph must be an unrooted undirected forest if we erase the direction of the edges. The third example in Figure 3 shows why acyclicity in the undirected sense is important.

To formally reason about ownership, we introduce the following functions:

$$\mathsf{out}(G, \nu) \in V \xrightarrow{\mathsf{fin}} L \qquad\qquad \mathsf{in}(G, \nu) \in \mathsf{Multiset}\ L$$

The outgoing edges $\mathsf{out}(G, \nu)$ determine which *resources vertex* ν *owns*, whereas the incoming edges $\mathsf{in}(G, \nu)$ determine at which labels (*i.e.*, types) the *vertex* ν *is owned*. We use the above functions in the definitions of the run-time typing judgment (Section 1.3.3) and the configuration well-formedness predicate (Section 1.3.4). We represent the the outgoing edges $\mathsf{out}(G, \nu)$ of a vertex ν as a finite map from vertices to labels to track *which* resources a vertex ν owns and at *which type*. The incoming edges $\mathsf{in}(G, \nu)$ of a vertex ν, however, we represent as a multiset of labels because it only matters at *which type* a vertex ν is owned, but not by whom (note that only channel endpoints can be owned).

WHAT IS PARAMETRIC IN THIS SECTION Connectivity graphs $\mathsf{Cgraph}(V, L)$ are parametric over the type of vertices V and labels L. All theory about connectivity graphs (including the separation logic) that we present throughout the rest of this section is parametric. Connectivity graphs and their theory are thus modularly separated from the operational semantics and type system of the language, which we found to be essential for keeping the complexity of the mechanization manageable.

1.3.3 *Run-Time Typing Judgment Using Separation Logic*

In the previous section, we developed the notion of a connectivity graph as a generalization of the heap typing, known from type safety proofs of the STLC with references (Harper, 2016; Pierce, 2002). We now make this generalization precise, develop a run-time typing judgment for our language, and show how we can use separation logic to hide reasoning about linearity.

We start with the run-time judgment $\Gamma; \Sigma \vdash e : \tau$, where $\Sigma \in V \xrightarrow{\mathsf{fin}} L$ provides the session types of the free channel references in e. Channel references amount to edges in our connectivity graph, and thus Σ becomes the set of *outgoing edges* $\mathsf{out}(G, \nu)$ associated with the threads and channels ν occurring in e. Let us consider the typing rule for channel references and function application:

$$\frac{\cdot}{\emptyset; \{\mathsf{Chan}(a) \mapsto (t, s)\} \vdash \#(a, t) : s} \qquad\qquad \frac{\Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2; \Sigma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2; \Sigma_1 \uplus \Sigma_2 \vdash e_1\ e_2 : \tau_2}$$

$$(\mathsf{Emp})(\Sigma) \triangleq (\Sigma = \emptyset) \qquad (P, Q \in \mathsf{iProp} \triangleq (V \xrightarrow{\mathsf{fin}} L) \to \mathsf{Prop})$$
$$(\ulcorner \phi \urcorner)(\Sigma) \triangleq \phi \wedge (\Sigma = \emptyset) \qquad (\Sigma \in V \xrightarrow{\mathsf{fin}} L)$$
$$(P \wedge Q)(\Sigma) \triangleq P(\Sigma) \wedge Q(\Sigma)$$
$$(\exists x.\, P(x))(\Sigma) \triangleq \exists x.\, P(x)(\Sigma)$$
$$(\mathsf{own}(\Sigma'))(\Sigma) \triangleq (\Sigma = \Sigma')$$
$$(P * Q)(\Sigma) \triangleq \exists \Sigma_1 \Sigma_2.\, \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2) = \emptyset \wedge \Sigma = \Sigma_1 \uplus \Sigma_2 \wedge P(\Sigma_1) \wedge Q(\Sigma_2)$$
$$(P \mathrel{-\!\!*} Q)(\Sigma) \triangleq \forall \Sigma'.\, \big(\mathsf{dom}(\Sigma) \cap \mathsf{dom}(\Sigma') = \emptyset \wedge P(\Sigma')\big) \Rightarrow Q(\Sigma \uplus \Sigma')$$

Figure 6: Semantics of separation logic.

Because our language is linear, we insist that the Σ-context is a singleton in the rule for channel references. In the rule for application, both contexts are split into disjoint parts for the subexpressions. Unfortunately, this leads to a multiplication of contexts and disjointness side conditions, because of the disjoint unions in the conclusion. These side conditions cannot be ignored because we want to mechanize our results. This is not a big issue for the variable context Γ since we mostly deal with *closed* expressions (*i.e.*, with $\Gamma = \emptyset$) because the operational semantics operates on closed expressions. The Σ-context, however, is in general non-empty for run-time expressions.

We use separation logic (O'Hearn et al., 2001) to hide the Σ-context and its disjointness conditions. We work with separation logic propositions $\mathsf{iProp} \triangleq (V \xrightarrow{\mathsf{fin}} L) \to \mathsf{Prop}$, which are predicates over a context $\Sigma \in V \xrightarrow{\mathsf{fin}} L$ of outgoing edges. Our use of separation logic as an *internal*, meta theoretical tool is inspired by Rouvoet et al. (2020) and contrasts with traditional uses which employ separation logic as an *external*, user-visible tool when specifying programs in Hoare-style logics. The separation logic connectives are defined in Figure 6. To assert that a separation logic proposition P is true, is to assert that $P(\emptyset)$ is true. An important special case is that $P \mathrel{-\!\!*} Q$ is true, if $\forall \Sigma.\, P(\Sigma) \Rightarrow Q(\Sigma)$.

Instead of the ordinary typing judgment $(\Gamma; \Sigma \vdash e : \tau) \in \mathsf{Prop}$ we define a separation-logic based judgment $(\Gamma \vdash e : \tau) \in \mathsf{iProp}$, so that $(\Gamma; \Sigma \vdash e : \tau)$ iff $(\Gamma \vdash e : \tau)(\Sigma)$. The preceding two rules are then reformulated as follows:

$$\frac{\mathsf{own}(\mathsf{Chan}(a) \mapsto (t, s))}{\emptyset \vdash \#(a, t) : s}* \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \quad * \quad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\ e_2 : \tau_2}*$$

The Σ-contexts are hidden by the separation logic connectives, and the disjointness conditions on Σ are taken care of by the separating conjunction ($*$). At a channel reference, we use the $\mathsf{own}(\Sigma)$ connective, which asserts that the separation logic resource is precisely Σ.

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

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

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad * \quad \Gamma_2 \uplus \{x \mapsto \tau_1\} \vdash e_2 : \tau_2}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}* \qquad \frac{\Gamma_1 \vdash e_1 : \mathbf{1} \quad * \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ () = e_1\ \mathsf{in}\ e_2 : \tau}*$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \quad * \quad \Gamma_2 \uplus \{x_1 \mapsto \tau_1\} \uplus \{x_2 \mapsto \tau_2\} \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{let}\ (x_1, x_2) = e_1\ \mathsf{in}\ e_2 : \tau}*$$

$$\frac{\Gamma_1 \vdash e_1 : \mathbf{N} \quad * \quad (\Gamma_2 \vdash e_2 : \tau \quad \wedge \quad \Gamma_2 \vdash e_3 : \tau)}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}* \qquad \frac{\Gamma_1 \vdash e_1 : !\tau.\, s \quad * \quad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{send}(e_1, e_2) : s}*$$

$$\frac{\Gamma \vdash e : ?\tau.\, s}{\Gamma \vdash \mathsf{receive}(e) : s \times \tau}* \qquad \frac{\Gamma \vdash e : \overline{s} \multimap 1}{\Gamma \vdash \mathsf{fork}(e) : s}* \qquad \frac{\Gamma \vdash e : \mathsf{End}}{\Gamma \vdash \mathsf{close}(e) : \mathbf{1}}* \qquad \frac{\mathsf{own}(\mathsf{Chan}(a) \mapsto (t, s))}{\emptyset \vdash \#(a, t) : s}*$$

Figure 7: The separation-logic based run-time type system of our language.

An exception to the rule that contexts are split up disjointly (with $*$) is the $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ expression. Although the channel references occurring in e_1 must be disjoint from those occurring in e_2 and e_3, the same endpoint is allowed to occur in both e_2 and e_3, because only one of the branches will be executed. This pattern too fits neatly in the separation logic methodology; we use separating conjunction ($*$) between e_1 and e_2, e_3, but ordinary conjunction ($\wedge$) between e_2 and e_3:

$$\frac{\Gamma_1 \vdash e_1 : \mathbf{N} \quad * \quad (\Gamma_2 \vdash e_2 : \tau \quad \wedge \quad \Gamma_2 \vdash e_3 : \tau)}{\Gamma_1 \uplus \Gamma_2 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}*$$

Figure 7 contains the full definition of our run-time type system using separation logic. Although it is possible to define the meaning of general inductive separation logic inference rules via Tarski's fixed point theorem, this generality is not necessary here: the expressions in the premises of each rule are strictly smaller than the expression in the conclusion, so the rules can be interpreted as being defined by recursion on the expression. We use this approach in the Coq formalization, because it has the additional benefit that we get the inversion rules for free.

A key strength of separation logic is that we can prove assertions using the proof rules of the logic of Bunched Implications (BI) (O'Hearn and Pym, 1999).

For example, separating conjunction ($*$) is associative and commutative, separating conjunction ($*$) has Emp as identity element, and magic wand ($-\!\!*$) is the adjoint of separating conjunction ($*$). We use the Iris Proof Mode (Krebbers et al., 2017, 2018) to reason abstractly using the rules of separation logic in Coq (see Section 1.7 for details).

WHAT IS PARAMETRIC IN THIS SECTION. The definition of the separation logic connectives and the proof rules for the separation logic are parametric in the types of vertices V and labels L.

1.3.4 *Well-Formedness of Configurations Using Connectivity Graphs*

Now that we have a run-time typing judgment for a single expression, we are in a position to define which configurations are well-formed. Recall that a configuration is a pair $(\vec{e}, h)$ where $\vec{e} : \mathsf{List}\ \mathsf{Expr}$ is the thread pool and where $h : \mathsf{Chan} \xrightarrow{\mathsf{fin}} \mathsf{List}\,\mathsf{Val}$ is the heap of channel buffers. We must certainly insist that all threads $\vec{e}$ are well-typed expressions (of unit type), and that all the values inside the heap h are well-typed. For the latter we have to ensure that a channel's endpoints are of dual types, modulo the messages queued up in the buffer. This requires us to consider the *incoming edges* $\mathsf{in}(G, v)$ of a vertex v in addition to its *outgoing edges* $\mathsf{out}(G, v)$. We can thus state well-formedness of a configuration in terms of its connectivity graph.

A configuration is well-formed if there exists a connectivity graph such that each thread and channel is locally well-formed with respect to its vertex in the graph:

$$\mathsf{wf}(\vec{e}, h) \triangleq \exists G : \mathsf{Cgraph}(V, L).\, \forall v \in V.\, \mathsf{wf}^{\mathit{local}}_{(\vec{e},h)}(v, \mathsf{in}(G, v))(\mathsf{out}(G, v))$$

Here, $\mathsf{wf}^{\mathit{local}}_{(\vec{e},h)} : V \times \mathsf{Multiset}\ L \to \mathsf{iProp}$ gives the *local well-formedness condition* for each vertex. It has two explicit arguments (the vertex $v \in V$ and its incoming edges $\mathsf{in}(G, v) \in \mathsf{Multiset}\ L$), and an extra argument $\mathsf{out}(G, v) \in V \xrightarrow{\mathsf{fin}} L$ that will form the vertex' local Σ-context, which is implicit in the type signature of $\mathsf{wf}^{\mathit{local}}$ because $\mathsf{iProp} \triangleq (V \xrightarrow{\mathsf{fin}} L) \to \mathsf{Prop}$.

A thread is locally well-formed if it is well-typed (with the implicit Σ-context given by its outgoing edges), and has no incoming edges (because threads cannot be owned):

$$\mathsf{wf}^{\mathit{local}}_{(\vec{e},h)}(\mathsf{Thread}(i), \Delta) \triangleq \begin{cases} \ulcorner \Delta = \emptyset \urcorner * \emptyset \vdash e_i : \mathbf{1} & \text{if } i < |\vec{e}| \\ \ulcorner \Delta = \emptyset \urcorner & \text{otherwise} \end{cases}$$

Note that wf quantifies over any vertex $v \in V$, and we thus have to consider any thread index i, including those $i \geqslant |\vec{e}|$ that are not yet in use. For such indexes, we use the separation logic proposition $\ulcorner \Delta = \emptyset \urcorner$ to assert that both the incoming and

outgoing edges are empty. The latter is implicit by the semantics of $\ulcorner \Delta = \emptyset \urcorner$ (see Figure 6).

A channel is locally well-formed if the buffers are well-typed (with the implicit Σ-context given by its outgoing edges), and have matching incoming edges match the types of the endpoints:

$$\mathsf{wf}^{\mathit{local}}_{(\vec{e},h)}(\mathsf{Chan}(a), \Delta) \triangleq \begin{cases} \exists s_0, s_1, s.\ \ulcorner \Delta = \{(0, s_0), (1, s_1)\} \urcorner * & \text{if } \#(a,0), \#(a,1) \in \mathsf{dom}(h) \\ \quad \vdash_{\mathit{buf}} h(a,0) : (s_0, s) * & \\ \quad \vdash_{\mathit{buf}} h(a,1) : (s_1, \overline{s}) & \\ \exists b, s.\ \ulcorner \Delta = \{(t, s)\} \urcorner * & \text{if } \#(a,t) \in \mathsf{dom}(h) \\ \quad \vdash_{\mathit{buf}} h(a,t) : (s, \mathsf{End}) & \quad \text{and } \#(a, 1-t) \notin \mathsf{dom}(h) \\ \ulcorner \Delta = \emptyset \urcorner & \text{otherwise} \end{cases}$$

In this definition we have to consider three cases. The first case corresponds to the situation in which both buffers are still in use. In that case, there must be two incoming edges in the connectivity graph, labeled with session types that are dual modulo the values in the buffers. For instance, if the left endpoint has session type $?\tau_1.?\tau_2.s$ and the right endpoint has session type $\overline{s}$, then the buffer of the left endpoint must be $[v_1, v_2]$ with $\vdash v_1 : \tau_1$ and $\vdash v_2 : \tau_2$. The second case corresponds to the situation in which one buffer has been deallocated. The third case corresponds to the situation in which the channel is not allocated (or both buffers have been deallocated).

The *buffer typing judgment* $\vdash_{\mathit{buf}} \vec{v} : (s_1, s_2)$ is inductively defined by the following rules:

$$\frac{}{\vdash_{\mathit{buf}} \epsilon : (s, s)}\ \mathsf{Emp}^* \qquad \frac{\emptyset \vdash v : \tau \quad * \quad \vdash_{\mathit{buf}} \vec{v} : (s_1, s_2)}{\vdash_{\mathit{buf}} ([v] +\!\!+ \vec{v}) : (?\tau.\, s_1, s_2)}\ *$$

These rules express that $\vdash_{\mathit{buf}} \vec{v} : (s_1, s_2)$ holds if s_1 is equal to prefixing s_2 with the types of the values $\vec{v}$ in the buffer. Note that similar to the other run-time judgments, the buffer typing judgment is defined in separation logic, which implicitly ensures that the Σ-environment is distributed disjointly over the values in the buffer.

WHAT IS PARAMETRIC IN THIS SECTION. The definition of wf is parametric in the type of vertices V and labels L, but also a *local well-formedness predicate* $\mathsf{wf}^{\mathit{local}}$ that captures the language-specific information by linking the incoming and outgoing edges of each vertex to their run-time counterpart.

1.3.5 *Proving Preservation Using Local Graph Transformations*

Now that we have defined the well-formedness predicate $\mathsf{wf}(\vec{e}, h)$, we must prove that it is preserved by the operational semantics: if $(\vec{e}, h) \rightsquigarrow_{\text{global}} (\vec{e}', h')$, then $\mathsf{wf}(\vec{e}, h)$

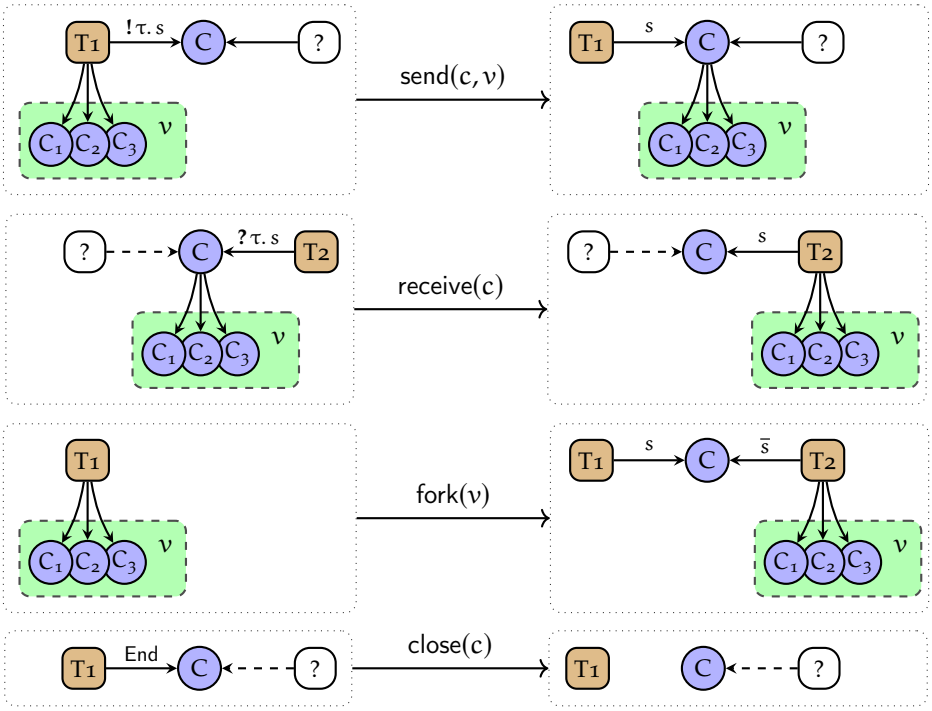


Figure 8: The operational steps and the corresponding connectivity graph transformations.

implies $\mathsf{wf}(\vec{e}', h')$. Recall that the well-formedness predicate $\mathsf{wf}(\vec{e}, h)$ intuitively means "there exists a connectivity graph G describing the configuration $(\vec{e}, h)$", so when the configuration steps to a new configuration $(\vec{e}', h')$, we must show that there exists a new connectivity graph G′ that describes $(\vec{e}', h')$.

If the head step is a pure step, then the new connectivity graph is exactly the same as the old one, and the preservation of the well-formedness predicate follows by a standard case analysis of the possible pure steps, because the heap does not change and no new threads are spawned.

Operational steps that involve channel operations are the interesting cases because they may alter the connectivity graph. Figure 8 provides a schematic overview. We focus on the $\mathsf{send}(c, v)$ operation, which moves the value v from the thread into the buffer of channel c. The session type in the label on the edge corresponding to c itself must change from $!\tau.\, s$ to s. Additionally, if the value v contains channel references, the connectivity graph must change to reflect this. The changes to the connectivity graph for send and the other channel operations are depicted in Figure 8.

Once we have chosen the appropriate new connectivity graph G′, we have to prove that this graph indeed describes the new configuration $(\vec{e}', h')$. This amounts to showing that the local well-formedness condition $\mathsf{wf}^{local}_{(\vec{e}', h')}(v, \mathsf{in}(G', v))(\mathsf{out}(G', v))$ is re-established for every vertex v.

For $\mathsf{send}(c, v)$ the parts of the $(\vec{e}, h)$-configuration and $(\vec{e}', h')$-configuration are:

$$e_i = K[\,\mathsf{send}(c, v)\,] \qquad h(\overline{c}) = \vec{v}$$
$$e'_i = K[\,c\,] \qquad h'(\overline{c}) = \vec{v} +\!\!+ [v]$$

The thread pool and heap do not change at other locations. After this change to the configuration and the corresponding change to the connectivity graph (as depicted in Figure 8), we classify the vertices into three types and explain how the local well-formedness $\mathsf{wf}^{local}_{(\vec{e}',h')}$ is restored.

1. For the vertices v' where neither the corresponding part of the configuration nor the incoming and outgoing edges change, $\mathsf{wf}^{local}_{(\vec{e},h)}(v', \mathsf{in}(G, v'))(\mathsf{out}(G, v'))$ remains valid.

2. For the vertices v' that correspond to channels referenced *inside the message* v, the owner changes from $\mathsf{Thread}(i)$ to $\mathsf{Chan}(c.1)$ (corresponding to T1 and C in the figure). These vertices are not affected either, because $\mathsf{in}(G, v') = \mathsf{in}(G', v')$. Since $\mathsf{in}(G, v')$ and $\mathsf{in}(G', v')$ are multisets of *labels*, they are thus not affected by the change of owner.

3. The vertices $v_1 = \mathsf{Thread}(i)$ and $v_2 = \mathsf{Chan}(c.1)$ (corresponding to T1 and C in the figure) are the vertices that are truly affected. Re-establishing their $\mathsf{wf}^{local}_{(\vec{e}',h')}(v_{12}, \mathsf{in}(G', v_{12}))(\mathsf{out}(G', v_{12}))$ requires some language-specific reasoning, because both their part of the configuration and their incoming and outgoing edges change.

There is another proof obligation that we need to meet: the connectivity graph has to remain acyclic when we do these local transformations.

Even though Figure 8 looks hopelessly language specific, we show that we can use our separation logic to distill abstract rules for *local graph transformations* (Section 1.5). These rules involve the transfer of resources between the old local well-formedness predicates $\mathsf{wf}^{local}_{(\vec{e},h)}(v, \mathsf{in}(G, v))$ and the new local well-formedness predicates $\mathsf{wf}^{local}_{(\vec{e}',h')}(v, \mathsf{in}(G', v))$ (for the affected vertices v in question). To distill these rules, it is crucial that the local well-formedness predicate is a separation logic proposition, which enables reasoning using the abstract proof rules of separation logic, without explicitly having to reference the graph, nor having to explicitly establish acyclicity, nor having to deal with disjointness conditions. The reasoning left to the user of the rule is purely local and precisely the language-specific reasoning that cannot be done generically. The result is that the preservation proof appears to perform no graph reasoning at all: at no point in the preservation proof is there any value of type $G, G' : \mathsf{Cgraph}(V, L)$ in the proof context.

WHAT IS PARAMETRIC IN THIS SECTION. The separation-logic based rules for *local graph transformations* (Section 1.5) are parametric in the type of vertices V and labels L, and the local well-formedness predicate.

1.3.6 *Proving Progress Using Waiting Induction*

To prove progress, we have to show that if $\mathsf{wf}(\vec{e}, h)$ holds, then either $\mathsf{final}(\vec{e}, h)$ holds (*i.e.*, $e_i = ()$ for all i and $h = \emptyset$), or the configuration can step. This is equivalent to saying that:

$$\mathsf{wf}(\vec{e}, h) \text{ and } \mathsf{active}(\vec{e}, h) \neq \emptyset \text{ implies that } (\vec{e}, h) \text{ can step}$$

Here, $\mathsf{active}(\vec{e}, h)$ is the set of threads and channels that have not yet terminated and not yet been fully deallocated, respectively.

Definition 1.3.3 (Active ⚙). The set of *active vertices* in configuration $(\vec{e}, h)$ is formally defined as

$$\mathsf{active}(\vec{e}, h) \triangleq \{\mathsf{Thread}(i) \mid e_i \neq ()\} \cup \{\mathsf{Chan}(a) \mid h(a, o) \neq \bot \vee h(a, 1) \neq \bot\}$$

If $\mathsf{active}(\vec{e}, h) \neq \emptyset$, then there exists a vertex $v \in \mathsf{active}(\vec{e}, h)$ for which we must find a thread that can step. If the vertex v is a thread that can step, we are done. The difficulty is that v may be a thread that is blocked on a $\mathsf{receive}(c)$, where the corresponding buffer of c in heap h is empty. If the configuration is well-formed, then we will presumably be able to find a non-blocked thread connected to the other endpoint of c, since that side will eventually be responsible for sending a message to c. However, the thread holding the other endpoint of c may be blocked itself, waiting on a receive on a different channel. Also, the endpoint c may not even be held by another thread; it could be stored in the buffer of some other channel c'.

Our way out is to use the connectivity graph: starting from vertex v, we search for another thread that can step. To organize this search process, we annotate edges of the connectivity graph with a *waiting direction* (as also done in Section 1.1), depicted as red triangles in Figure 9. The waiting direction is formalized using the notion of being blocked.

Definition 1.3.4 (Blocked ⚙). A vertex v_1 is *blocked* on vertex v_2 in configuration $(\vec{e}, h)$ if v_1 is a thread that is trying to receive from channel v_2 whose buffer is empty. Formally:

$$\mathsf{blocked}_{(\vec{e},h)}(v_1, v_2) \triangleq \exists i, a, t, K.\ v_1 = \mathsf{Thread}(i) \wedge v_2 = \mathsf{Chan}(a) \wedge e_i = K[\,\mathsf{receive}(\#(a, t))\,] \wedge h(a, t) = \epsilon$$

The waiting direction (red triangle) $v_1 \to^{l}_{G} v_2$ of an edge coincides with its ownership direction (black arrowhead) if v_1 is blocked on v_2. Otherwise, it is opposite to the ownership direction.

To find a thread that can step from $v \in \mathsf{active}(\vec{e}, h)$, we follow edges in the waiting direction until we arrive at a vertex that has no outgoing waiting edges. As one can see in Figure 9, if we follow the waiting direction (red triangles) from *any* start vertex v, we always end up in a thread that can step (green dotted square). To prove that

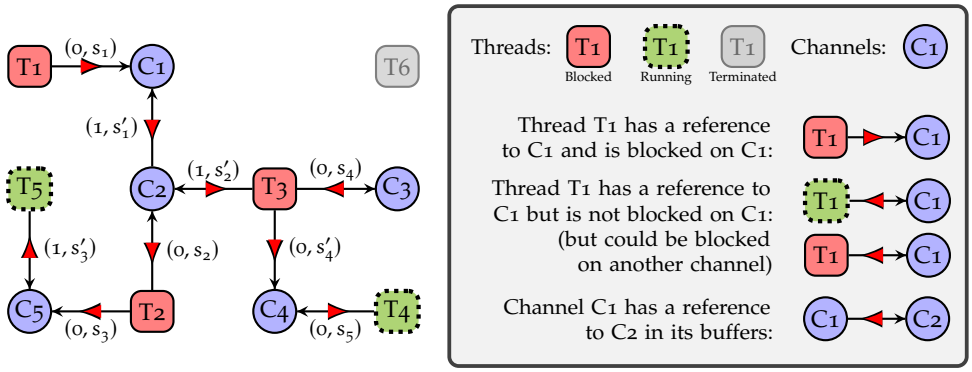


Figure 9: The connectivity graph from Figure 5 annotated with red triangles for the waiting direction.

we can always find a thread that can step by simply following the waiting edges from any starting vertex, we have to show that:

1. If the current vertex v is a thread, it can either step, or it has an outgoing waiting edge.
2. If the current vertex v is a channel, it always has an outgoing waiting edge.
3. The search process terminates, because the graph is acyclic as an undirected graph.

TO SHOW (1): We show that active threads v can step or have an outgoing waiting arrow by induction on typing. The interesting cases are the channel operations, and receive in particular, so suppose that the thread's expression is $K[\,\mathsf{receive}(v)\,]$. By run-time typing, we know that v is a channel reference $\#(a, t)$, and the typing rule for $\mathsf{receive}(v)$ gives us the separation logic resource $\mathsf{own}(\{\mathsf{Chan}(a) \mapsto (t, ?\,\tau.\, s)\})$. From this it follows that the thread has an outgoing edge to $\mathsf{Chan}(a)$, and hence $\mathsf{Chan}(a)$ has an incoming edge with the label $(t, ?\,\tau.\, s)$. From the channel's local well-formedness predicate, it follows that the required buffer exists in the heap. If the buffer is non-empty, then the receive can proceed, so the configuration can step. If the buffer is empty, we have an outgoing waiting arrow from the thread to the channel, so the search process can continue.

TO SHOW (2): The channel v is active, so it has a buffer, so it must have a corresponding incoming edge in the graph by the definition of the local well-formedness predicate for channels. If that incoming edge comes from a vertex v', and that vertex v' is not blocked on v, we are done. That is because then the waiting direction is pointing from v to v', and we can continue the search process from v'. If v' is a thread currently blocked on v, then the session type on that edge must be a receive. It follows from the channel's local well-formedness predicate that

the other endpoint has not yet been closed, and thus there is another incoming edge. It cannot be the case that both buffers are empty and the other edge is also a receive, because that would violate duality. Thus, the other edge is coming from a vertex v'' that is *not* a thread currently blocked on us. So there is a waiting edge from v to v'', and we can continue the search process from v''.

TO SHOW (3): Although (3) is intuitively obvious if one looks at a picture such as Figure 9, one has two difficulties in a formal setting. Firstly, showing that such a search process actually terminates requires formally reasoning about the (undirected) acyclicity of graphs. We refer the interested reader to our appendix and Coq mechanization for details (Jacobs et al., 2021). Secondly, one has to *restructure the argument* in order to even formally state what it means that "the search process terminates". Our key idea is that the progress proof can be proved with an *inductive argument*, with a non-standard graph induction principle.

We call this induction principle for connectivity graphs *waiting induction*. The induction principle says that in order to prove $P(v)$ for all vertices $v \in V$, we can assume that $P(v')$ already holds for all vertices v' that v is waiting for. Note the similarity with strong induction on natural numbers: in order to prove $P(n)$ for all $n \in \mathbb{N}$, we can assume that $P(n')$ already holds for all $n' < n$.

We restructure the progress proof by applying our waiting induction principle at the start. Whenever we said "continue the search process" in the argument above, we can apply the inductive hypothesis. The induction principle is formally stated and discussed in more detail in Section 1.4.

WHAT IS PARAMETRIC IN THIS SECTION. The waiting induction principle is parametric in the types of vertices V and labels L. This induction principle encapsulates the acyclicity reasoning, so that the progress proof can focus on the language-specific reasoning.

1.4 CONNECTIVITY GRAPHS AND WAITING INDUCTION IN DETAIL

Reasoning about graphs in a progress and preservation proof is non-standard, and reasoning about graphs in a proof assistant is more involved than reasoning about inductively-defined types like lists or maps that are normally used to define heap typings. We therefore factor graph reasoning out into a connectivity graph *library* that is *parametric* in vertices V and labels L. In this section we explain the foundations of this library by presenting the formal definition of acyclicity, a selected set of primitive rules (which are used to prove soundness of our separation-logic based graph transformations in Section 1.5 for proving preservation), as well as our principle of waiting induction (for proving progress). We hope to convince the reader that our graph-based approach is feasible—even in a mechanized setting in a proof assistant.

Recall the informal definition of connectivity graphs Cgraph from Section 1.3.2:

$$\mathsf{Cgraph}(V, L) \triangleq \left\{ G \in V \times V \xrightarrow{\text{fin}} L \;\middle|\; G \text{ has no undirected cycles} \right\}$$

To define "G has no undirected cycles" formally, we need to introduce some basic notions about graphs. We let $\mathsf{graph}(V, L) \triangleq V \times V \xrightarrow{\text{fin}} L$ be graphs without the acyclicity restriction. The notation $v_1 \to^{l}_{G} v_2$ expresses that there is an edge from vertex v_1 to v_2 with label l (*i.e.*, we have $G(v_1, v_2) = l$). The notation $v_1 \leftrightarrow_G v_2$ expresses that there is an edge from v_1 to v_2 or from v_2 to v_1. The notation $v_1 \leftrightarrow^{*}_{G} v_2$ expresses that vertices v_1 and v_2 are *connected* by a (possibly empty) path from v_1 to v_2 where we may follow edges in either direction, and $v_1 \not\leftrightarrow^{*}_{G} v_2$ expresses that there is no such path.

Definition 1.4.1 (Undirected acyclicity ⚙)**.** A graph $G \in \mathsf{graph}(V, L)$ has no undirected cycles if:

1. The undirected erasure $\bar{G} = \{\{v_1, v_2\} \mid v_1 \leftrightarrow_G v_2\}$, where we forget the labels and directions of the edges, is acyclic. See Jacobs et al. (2021) for details about the formalization of acyclicity of undirected graphs and the undirected erasure.
2. There are no short loops, *i.e.*, we do not both have $v_1 \to^{l}_{G} v_2$ and $v_2 \to^{l'}_{G} v_1$.

Our reasoning about the acyclicity of graphs rests on two primitive lemmas:

Lemma 1.4.2 (Graph insertion ⚙)**.** *If* $G \in \mathsf{graph}(V, L)$ *is a graph with no undirected cycles and* $v_1 \not\leftrightarrow^{*}_{G} v_2$*, then* $G \cup \{v_1 \to^{l} v_2\}$ *has no undirected cycles.*

Lemma 1.4.3 (Graph deletion ⚙)**.** *If* $G \in \mathsf{graph}(V, L)$ *is a graph with no undirected cycles and* $v_1 \to^{l}_{G} v_2$*, then* $v_1 \not\leftrightarrow^{*}_{H} v_2$ *in the graph* $H \triangleq G \setminus \{v_1 \to^{l} v_2\}$.

We build a library of derived lemmas on top of these two primitive lemmas. A lemma that is crucial for proving the correctness of our separation logic rules in Section 1.5 is the exchange lemma, which is used to exchange separation logic resources between vertices of the graph:

Lemma 1.4.4 (Graph exchange ⚙)**.** *Let* $G, H \in \mathsf{graph}(V, L)$ *be graphs and let* $v_1, v_2 \in V$ *be vertices. If*

1. G *has no undirected cycles,*
2. $v_1 \not\leftrightarrow^{*}_{G} v_2$*,*
3. $\mathsf{out}(G, v_1) \uplus \mathsf{out}(G, v_2) = \mathsf{out}(H, v_1) \uplus \mathsf{out}(H, v_2)$*, and*
4. $\mathsf{out}(G, v) = \mathsf{out}(H, v)$ *for all* $v \in V \setminus \{v_1, v_2\}$.

Then:

1. H *has no undirected cycles,*

2. $v_1 \not\leftrightarrow^*_H v_2$, *and*

3. $\mathsf{in}(G, v) = \mathsf{in}(H, v)$ *for all* $v \in V$.

This lemma is quite a mouthful, so let us go over it step by step. We start with a graph G and we want to exchange outgoing edges between two unconnected vertices v_1 and v_2 to obtain a graph H in which the union of the outgoing edges of v_1 and v_2 stays the same. The lemma tells us that this operation maintains undirected acyclicity and that v_1 and v_2 are unconnected. Furthermore, the labels of incoming edges stay the same for all vertices.

Note that this property only holds because $\mathsf{in}(G, v)$ is a multiset rather than a map that stores the vertices, like we did for $\mathsf{out}(G, v)$. The fact that the local invariants are unaware of the vertices of origin of the incoming edges is what enables local reasoning: exchange of edges only affects the local invariants of v_1 and v_2. In particular, for a channel it does not matter if its owner changes due to an exchange of resources, because it only matters at which type the channel is owned.

A typical pattern is to compose the lemma for exchange with with lemma for insertion and deletion. For instance, given an edge $v_1 \to^l_G v_2$, we can first delete the edge using Theorem 1.4.2 to obtain $v_1 \not\leftrightarrow^*_H v_2$. Then we can apply Theorem 1.4.4 to exchange some of the outgoing edges of v_1 and v_2, and then we can re-insert a new edge $v_1 \to^{l'} v_2$ with a new label l using Theorem 1.4.2.

The lemmas for insertion and deletion (Theorems 1.4.2 and 1.4.3) can not only be used to prove the acyclicity of modified connectivity graphs, but also to prove structural properties of connectivity graphs. The simplest example is a lemma that connectivity graphs have no self loops, which we give here as an illustration that the lemmas for insertion and deletion suffice.[6]

Lemma 1.4.5 (No self loops ⚙). *A connectivity graph* $G \in \mathsf{Cgraph}(V, L)$ *has no self loops* $v \to^l_G v$.

Proof. Suppose that $v \to^l_G v$. By Theorem 1.4.3, $v \not\leftrightarrow^*_{G'} v$ in the connectivity graph $G' \triangleq G \setminus \{v \to^l v\}$. Since every vertex is connected to itself (by definition), we have a contradiction. □

Another example of a structural property that follows from the lemmas for insertion and deletion is the separation lemma. In Section 1.5 this lemma will play an important role in enabling our use of separation logic, where the separating conjunction requires that resources are disjoint.

Lemma 1.4.6 (Separation ⚙). *If* $G \in \mathsf{Cgraph}(V, L)$ *and* $v_1 \not\leftrightarrow^*_G v_2$ *or* $v_1 \leftrightarrow_G v_2$, *then the outgoing edges of* v_1 *and* v_2 *are disjoint, i.e.,* $\mathsf{dom}(\mathsf{out}(G, v_1)) \cap \mathsf{dom}(\mathsf{out}(G, v_2)) = \emptyset$.

Lastly, we have our generic principle of waiting induction that is key to our progress proof.

6 We do actually need this lemma at various points in the Coq proofs of the lemmas in Section 1.5.

Lemma 1.4.7 (Waiting induction ⚙). *Let* $G \in \mathsf{Cgraph}(V, L)$ *be a connectivity graph,* $P \in V \to \mathsf{Prop}$ *a predicate over* V*, and* $R : V \times V \to \mathsf{Prop}$ *a binary relation over* V*.*

Then in order to prove $\forall v \in V.\, P(v)$ *it suffices to prove:*

$$\begin{aligned}\forall v \in V.\, \big(\forall v' \in V.\, (v \to^{l}_{G} v' \wedge R(v, v')) \Rightarrow P(v')\big) \Rightarrow \\ \big(\forall v' \in V.\, (v' \to^{l}_{G} v \wedge \neg R(v', v)) \Rightarrow P(v')\big) \Rightarrow P(v)\end{aligned}$$

In other words, to prove $P(v)$, we can assume that $P(v')$ already holds for outgoing neighbors v' of v that are in relation $R(v, v')$, and we can also assume that $P(v')$ holds for incoming neighbors v' of v that are not in relation $R(v', v)$.

Thus, for neighbors $v \to^{l} v'$, either the proof of $P(v)$ can assume $P(v')$, or *vice versa*, but not both, and the relation $R(v, v')$ determines which. This induction principle is well founded due to the acyclicity of connectivity graphs. We prove this lemma using a similar lemma for undirected graphs, which we detail in Jacobs et al. (2021).

We call the lemma *waiting induction* because (1) we choose $R \triangleq \mathsf{blocked}_{(\vec{e},h)}$ from Section 1.3.6 and thus R is the waiting relation in our application, and (2) because of the similarity to induction on natural numbers: in order to prove $P(n)$ we can assume that $P(n-1)$ already holds, if $n \neq 0$.

1.5 LOCAL GRAPH TRANSFORMATION RULES IN SEPARATION LOGIC

We now generalize the well-formedness predicate wf from Section 1.3.4 to become parametric in the vertices V and labels L, which involves making it parametric in the local well-formedness predicate to abstract from language-specific aspects. We state separation-logic based rules for the parametric well-formedness predicate so that preservation can be proved using local reasoning. After an initial attempt at a monolithic proof of preservation, we found our approach of separating the graph reasoning from the local language-specific reasoning to be indispensable for mechanization.

Given a local well-formedness predicate $P : V \times \mathsf{Multiset}\, L \to \mathsf{iProp}$, we define the generic global well-formedness predicate $\mathsf{wf}(P)$ as follows:

$$\mathsf{wf}(P) \triangleq \exists G : \mathsf{Cgraph}(V, L).\, \forall v \in V.\, P(v, \mathsf{in}(G, v))(\mathsf{out}(G, v))$$

We can instantiate the above definition with $P \triangleq \mathsf{wf}^{local}_{(\vec{e},h)}$ to obtain the well-formedness predicate from Section 1.3.4 that was tied to our concrete language.

Recall from Section 1.3.5 that preservation means: if $(\vec{e}, h) \rightsquigarrow_{\mathrm{global}} (\vec{e}', h')$, then $\mathsf{wf}(\mathsf{wf}^{local}_{(\vec{e},h)})$ implies $\mathsf{wf}(\mathsf{wf}^{local}_{(\vec{e}',h')})$. We now present a set of *graph transformation rules* for proving results "$\mathsf{wf}(P)$ implies $\mathsf{wf}(P')$" where P and P' are arbitrary local well-formedness predicates, instead of a concrete local well-formedness predicate. These graph transformation rules perform a transformation of the graph under the hood, but the graphs are encapsulated by the definition of wf, and the rules thus do not

mention any graphs. Instead, the premises of these graph transformation rules ask the user of the rule to prove *local* separation logic entailments involving P and P'.

The first of these graph transformation rules allows the user to exchange separation logic resources between two vertices $v_1, v_2 \in V$ in order to prove that $\mathsf{wf}(P)$ implies $\mathsf{wf}(P')$:

Lemma 1.5.1 (Exchange ⚙). *Let* $v_1, v_2 \in V$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*[7]

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

This rule generalizes the transformations for send and receive from Figure 8 where resources are exchanged between two vertices. We go over the premises of the rule in detail:

1. The first premise asks the user of the rule to prove the local implication $P(v, \Delta) \mathrel{-\!\!*} P'(v, \Delta)$ for the vertices $v \in V \setminus \{v_1, v_2\}$ that are not involved in the exchange.
2. The second premise first gives the user access to the local resources $P(v_1, \Delta_1)$ of vertex v_1. The rule then asks the user to prove that there exists an edge $v_1 \to^{l}_{G} v_2$, by showing that $\exists l.\, \mathsf{own}(v_2 \mapsto l) * \cdots$ follows from the local resources of v_1. The rule then gives the user access to the local resources $P(v_2, \{l\} \uplus \Delta_2)$ of vertex v_2, where we have obtained the information that the label $\{l\}$ is part of the incoming edge label multiset of v_2. The rule then allows the user to pick a new label l' for the edge $v_1 \to^{l'} v_2$. Subsequently, the user has to restore the local resources of v_1 and v_2 for the new P'. For restoring the local resources $P'(v_1, \Delta_1)$, the user additionally gets the $\mathsf{own}(v_2 \mapsto l')$ of the new edge. For restoring the local resources $P'(v_2, \{l'\} \uplus \Delta_2)$, we get the new label in the incoming edge label multiset.

It may seem like this rule only allows us to change the label on the edge $v_1 \to^{l}_{G} v_2$ from l to l', but the rule in fact allows us to arbitrarily exchange separation logic resources (*i.e.,* outgoing edges) between v_1 and v_2. The way this works is that the rule gives us access to the old local resources of both v_1 and v_2, and it asks us to prove the separating conjunction of the new local resources of both v_1 and v_2. The proof rules of separation logic allow us to use resources stored in the old local resources of v_1 to prove the new local resources of v_2, and *vice versa*. Thus, the graph transformation that is applied internally in the rule depends on *which* proof of the separation logic entailment is provided by the user of the rule.

7 Recall that proving $P \in \mathsf{iProp}$ means proving $P(\emptyset)$ (see Section 1.3.3), but in practice (and in Coq) this is done using the proof rules of separation logic.

A NOTE ON THE PROOF OF THE TRANSFORMATION RULE. That the transformation rule is able to offer us access to both local resources simultaneously relies crucially on the acyclicity of the graph. The acyclicity, and the existence of an edge between the two vertices, is what allows us to apply the separation lemma (Theorem 1.4.6) that allows us to construct the separating conjunction of the two local resources. In the proof of the rule we re-establish the validity of the resources and the acyclicity of the graph using the exchange lemma (Theorem 1.4.4).

In addition to the preceding transformation rule for changing the label on an edge (and exchanging resources), we have a transformation rule for removing an edge (after exchanging resources). This rule is used in the close case of the preservation proof:

Lemma 1.5.2 (Deallocation ⚙). *Let* $v_1, v_2 \in \mathsf{V}$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*

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

We have the following two transformation rules for inserting an outgoing/incoming edge between v_1 and v_2, respectively. To maintain acyclicity, we have to show that v_2 has no existing outgoing edges. Like the preceding rules, these rules also allow us to transfer resources to v_2.

Lemma 1.5.3 (Allocation out ⚙). *Let* $v_1, v_2 \in \mathsf{V}$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*

1. $P(v, \Delta) \dashv\!\!* P'(v, \Delta)$ *for all* $v \in \mathsf{V} \setminus \{v_1, v_2\}$ *and* $\Delta \in \mathsf{Multiset}\ \mathsf{L}$
2. $P(v_2, \Delta_2) \dashv\!\!* \ulcorner \Delta_2 = \emptyset \urcorner$ *for all* $\Delta_2 \in \mathsf{Multiset}\ \mathsf{L}$
3. $P(v_1, \Delta_1) \dashv\!\!* \exists l'.\, (\mathsf{own}(v_2 \mapsto l') \dashv\!\!* P'(v_1, \Delta_1)) * P'(v_2, \{l'\})$ *for all* $\Delta_1 \in \mathsf{Multiset}\ \mathsf{L}$

Lemma 1.5.4 (Allocation in ⚙). *Let* $v_1, v_2 \in \mathsf{V}$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*

1. $P(v, \Delta) \dashv\!\!* P'(v, \Delta)$ *for all* $v \in \mathsf{V} \setminus \{v_1, v_2\}$ *and* $\Delta \in \mathsf{Multiset}\ \mathsf{L}$
2. $P(v_2, \Delta_2) \dashv\!\!* \ulcorner \Delta_2 = \emptyset \urcorner$ *for all* $\Delta_2 \in \mathsf{Multiset}\ \mathsf{L}$
3. $P(v_1, \Delta_1) \dashv\!\!* \exists l'.\, P'(v_1, \Delta_1 \uplus \{l'\}) * (\mathsf{own}(v_1 \mapsto l') \dashv\!\!* P'(v_2, \emptyset))$ *for all* $\Delta_1 \in \mathsf{Multiset}\ \mathsf{L}$

Lastly, we have a derived transformation rule that adds two new edges $v_1 \rightarrow^{l_1'} v_2$ and $v_2 \leftarrow^{l_2'} v_3$. We use this rule in the fork case of the preservation proof.

Lemma 1.5.5 (Allocation out and in ⚙). *Let* $v_1, v_2, v_3 \in \mathsf{V}$. *To prove* $\mathsf{wf}(P)$ *implies* $\mathsf{wf}(P')$, *it suffices to prove:*

1. $P(v, \Delta) \dashv\!\!* P'(v, \Delta)$ *for all* $v \in \mathsf{V} \setminus \{v_1, v_2, v_3\}$ *and* $\Delta \in \mathsf{Multiset}\ \mathsf{L}$

$$\frac{\Gamma \text{ unrestricted}}{\{x \mapsto \tau\} \uplus \Gamma \vdash x : \tau} \qquad \frac{n \in \mathbb{N} \qquad \Gamma \text{ unrestricted}}{\Gamma \vdash n : \mathbf{N}}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 \cup \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2 \qquad \Gamma \text{ unrestricted}}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma_1 \perp \Gamma_2 \qquad \Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1 \cup \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

Figure 10: Selected typing rules for unrestricted types.

2. $P(\nu, \Delta) \twoheadrightarrow \ulcorner \Delta = \emptyset \urcorner$ *for all* $\nu \in \{\nu_2, \nu_3\}$ *and* $\Delta \in \mathsf{Multiset}\ L$
3. $P(\nu_1, \Delta_1) \twoheadrightarrow \exists l_1', l_2'.\ (\mathsf{own}(\nu_2 \mapsto l_1') \twoheadrightarrow P'(\nu_1, \Delta_1 \uplus \{l'\})) * P'(\nu_2, \{l_1', l_2'\}) * (\mathsf{own}(\nu_2 \mapsto l_2') \twoheadrightarrow P'(\nu_3, \emptyset))$
   for all $\Delta_1 \in \mathsf{Multiset}\ L$

This rule can be proved by applying both allocation out and allocation in. It pays off to prove this in the generic setting, because the intermediate state (in which the channel has been allocated but not yet the thread that will hold the other endpoint) is not well-formed according to our wf^{local}. Instead, we prove $\mathsf{wf}(P) \implies \mathsf{wf}(P')$ by carefully choosing Q and proving $\mathsf{wf}(P) \implies \mathsf{wf}(Q)$ using Theorem 1.5.3, and $\mathsf{wf}(Q) \implies \mathsf{wf}(P')$ using Theorem 1.5.4.

1.6 EXTENSIONS

The programming language for which we have mechanized deadlock and memory leak freedom in Coq (Jacobs et al., 2021) supports more features than described in Section 1.2. First, it has more standard features such as sum types, which we do not describe because their rules are standard and the modification of the proof is straightforward. Second, it has unrestricted (non-linear) types, including unrestricted products and sums, and an unrestricted function type (Section 1.6.1) and general equi-recursive functional types (which can encode algebraic data types) and equi-recursive recursive session types (which can encode infinite protocols) (Section 1.6.2). Furthermore, we prove a deadlock freedom property that is stronger than global progress and also rules out partial deadlock (Section 1.6.3).

1.6.1 *Unrestricted Types*

We make the types used for conventional functional programming (such as product, sum, and function types) unrestricted (*i.e.*, non-linear) if their components are unrestricted. Instead of introducing separate linear and non-linear products and

sums, we introduce the judgment "τ unrestricted" on types τ, which holds if all the components of τ are unrestricted. Formally, the base types $\mathbf{0}$, $\mathbf{1}$ and $\mathbf{N}$ are unrestricted, and $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are unrestricted if τ_1 and τ_2 are unrestricted. The type $\tau_1 \multimap \tau_2$ is always linear (*i.e.*, not unrestricted), even if τ_1 and τ_2 are unrestricted, because the closure may capture linear variables. We introduce the type $\tau_1 \to \tau_2$ of unrestricted functions, which is always unrestricted (even if τ_1 and τ_2 are linear), and whose closures are only allowed to capture unrestricted variables. Selected typing rules are shown in Figure 10. The typing rules involve the *disjointness relation* $\Gamma_1 \perp \Gamma_2$, which expresses that Γ_1 and Γ_2 might share unrestricted variables, but otherwise do not overlap. Formally:

$$\Gamma_1 \perp \Gamma_2 \triangleq \forall x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2).\, \Gamma_1(x) = \Gamma_2(x) \wedge \Gamma_1(x) \text{ unrestricted}$$

CHANGES TO THE PROOF. In order to reason about unrestricted values in the separation logic, we add a standard box modality $\Box P$, defined as $(\Box P)(\Sigma) \triangleq P(\emptyset) \wedge \Sigma = \emptyset$. The box modality asserts that P does not use any linear resources, which allows it to support proof rules for deletion ($\Box P \mathrel{-\!\!*} \mathsf{Emp}$) and duplication ($\Box P \mathrel{-\!\!*} \Box P * \Box P$). Lastly, we have the rule $\Box P \mathrel{-\!\!*} P$ to open the box. We use the box modality in the run-time typing rule for the unrestricted function type:

$$\frac{\Box\,(\Gamma \uplus \{x \mapsto \tau_1\} \vdash e : \tau_2) \quad * \quad \ulcorner \Gamma \text{ unrestricted} \urcorner}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}*$$

The box modality makes sure that the closure cannot capture any channels at run-time.

We prove $(\Gamma \vdash e : \tau) \mathrel{-\!\!*} \Box\,(\Gamma \vdash e : \tau)$ if τ unrestricted. This entailment says that run-time typing judgments for expressions e of unrestricted type τ can be freely deleted and duplicated in the separation logic sense. This is crucial for the main change to our proof—the substitution lemma—in which we now have to consider the case that the type is unrestricted, and that a variable could be substituted in multiple or zero places. We use the preceding entailment and the laws of the box modality to adapt the proof of the substitution lemma.

1.6.2 *Equi-Recursive Types*

We extend our type system with equi-recursive functional ($\mu\alpha.\tau$) and session types ($\mu\alpha.s$), in order to be able to encode algebraic data types and infinite protocols, respectively. We extend the type system with the following rule for unfolding recursive types:

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 \equiv \tau_2}{\Gamma \vdash e : \tau_2}$$

The congruence relation ($\equiv$) relates types up to unfolding of $\mu\alpha.\tau \equiv \tau[\mu\alpha.\tau/\alpha]$. Our mechanization (Section 1.7) is somewhat more general: we use a coinductive definition of types to allow mutual recursion and recursion through the message type as well as the tail. We also extend unrestricted types to allow recursive types to be unrestricted. We can encode algebraic data types such as lists by using sums and products and recursive types.

CHANGES TO THE PROOF. We do *not* add a rule for unfolding recursive types to the run-time type system. Rather, we define the run-time type system in a syntax directed way so that all constructors respect the congruence relation ($\equiv$), and then *prove* a version of the unfolding rule:

Lemma 1.6.1. *If* $\Gamma_1 \equiv \Gamma_2$ *and* $\tau_1 \equiv \tau_2$, *then* $(\Gamma_1 \vdash e : \tau_1) \nrightarrow (\Gamma_2 \vdash e : \tau_2)$.

EXAMPLE. The combination of equi-recursive and unrestricted types allows us to type check the call-by-value Y-combinator for constructing recursive functions of type $\tau_1 \to \tau_2$. Defining recursive functions in terms of a self-referential type is standard (Harper, 2016):

$$Y : ((\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)) \to (\tau_1 \to \tau_2)$$
$$Y \triangleq \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

We use the recursive type $\mu\alpha.(\alpha \to (\tau_1 \to \tau_2))$ for x and the type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$ for f. Note that while τ_1 and τ_2 can be restricted (linear) types, we *must* use an unrestricted function type for f and x in order to type check the multiple uses of f and x. In fact, a fixed-point combinator for constructing functions $\tau_1 \multimap \tau_2$ with linear function type would violate type safety. Intuitively, a recursive function is allowed to *manipulate* both linear and non-linear resources, but the *definition* of a recursive function is not allowed to *capture* linear resources in its closure because this closure will be invoked multiple times.

1.6.3 *Partial Deadlock and Memory Leak Freedom via Reachability*

In the context of session-typed languages with non-termination (*e.g.*, due to recursive types), deadlock freedom is typically stated as *global progress*, which we prove in Section 1.3.6. Global progress guarantees that the configuration can either take a step, or is in a final state where all threads have successfully terminated and all channels have been deallocated. Although global progress rules out whole-program deadlocks, as well as memory leaks when all threads have terminated, it admits partial deadlocks as long there is still a thread that can step (*e.g.*, is in an infinite loop). Linear session types actually rule out partial deadlocks and memory leaks even when some threads are still running. Although deadlock freedom and memory leak freedom may seem like separate properties, we state two properties that

simultaneously generalize both, namely *partial deadlock/leak freedom* (Theorem 1.6.4) and *full reachability* (Theorem 1.6.6). We prove that these properties are equivalent (Theorem 1.6.7) and show that full reachability can be proven using the waiting induction principle of our proof method (Theorem 1.6.8). Finally, we show that they imply global progress.

In order to arrive at a simultaneous generalization of deadlock and memory leak freedom, consider pure memory leaks and pure deadlocks:

- A *pure memory leak* is one in which we have a set S of channels, such that all endpoints of the channels in S are held by the buffers of channels in the same set S.
- A *pure deadlock* is a set S of both threads and channels with empty buffers, such that all threads in S are blocked on one of the channels in the set S, and all of the endpoints of the channels in S are held by threads in the set S.

In general, we can have a mixed partial deadlock/leak situation that is neither a pure memory leak nor a pure deadlock. Intuitively, a partial deadlock and memory leak is a set S of threads and channels such that all threads in S are blocked on one of the channels in S, and all endpoints of channels in S are held by threads and channels in S. To make this formal, we define the set of vertices $\mathsf{refs}_{(\vec{e},h)}(v) \subseteq V$ that a vertex v references.

Definition 1.6.2. We let $\mathsf{refs}_{(\vec{e},h)}(\mathsf{Thread}(i)) \triangleq \{\mathsf{Chan}(a') \mid \text{channel literal } \#(a', t) \text{ occurs in } e_i\}$, and $\mathsf{refs}_{(\vec{e},h)}(\mathsf{Chan}(a)) \triangleq \{\mathsf{Chan}(a') \mid \text{channel literal } \#(a', t) \text{ occurs in } h(\#(a, 0)) \text{ or } h(\#(a, 1))\}$.

With this function at hand, we can define partial deadlock and memory leak freedom.

Definition 1.6.3 (Partial deadlock/leak). **Given a configuration $(\vec{e}, h)$, a subset $S \subseteq V$ of the threads and channels is in a partial *deadlock/leak* if the following conditions hold:**

1. We have $\emptyset \subset S \subseteq \mathsf{active}(\vec{e}, h)$ (see Theorem 1.3.3 for the definition of active).
2. For all threads $\mathsf{Thread}(i) \in S$, the expression e_i cannot step in the heap h.
3. If $\mathsf{Thread}(i) \in S$ and $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Thread}(i), \mathsf{Chan}(a))$, then $\mathsf{Chan}(a) \in S$ (see Theorem 1.3.4 for the definition of blocked).
4. If $\mathsf{Chan}(a) \in S$ and $\mathsf{Chan}(a) \in \mathsf{refs}_{(\vec{e},h)}(v)$, then $v \in S$.

Definition 1.6.4 (Partial deadlock/leak freedom). **A configuration $(\vec{e}, h)$ is *deadlock/leak free* if no $S \subseteq V$ is in a partial deadlock/leak in $(\vec{e}, h)$.**

In order to prove that well-formed configurations have no partial deadlock/leak, we prove another property that we call *full reachability*, which we show to be equivalent to partial deadlock/leak freedom. Full reachability has the advantage

that it can be proved directly using waiting induction. It takes inspiration from the notion of reachability used in garbage collection and memory management, namely that data is said to be reachable if it can be reached by transitively following pointers, starting from any thread's stack frames. Memory leak freedom can then be stated as: all data in the configuration is reachable, *i.e.,* there is never any leaked memory. To incorporate deadlock freedom into this, we strengthen the definition of reachability to only start from stack frames of *threads that can step*. However, if a thread T1 is blocked on channel C, and the other endpoint of C is held by still running thread T2, then data held by T1 should also be considered transitively reachable: even though this data is held by a thread that (currently) cannot step, further interaction of T2 with the channel C may unblock T1. We formalize this using the following inductive definition:

Definition 1.6.5 (Reachability ⚙)**.** We inductively define the vertices that are *reachable* in $(\vec{e}, h)$:

1. $\mathsf{Thread}(i)$ is reachable if either
   - the expression e_i can step in the heap h, or
   - there exists an α such that $\mathsf{blocked}_{(\vec{e},h)}(\mathsf{Thread}(i), \mathsf{Chan}(\alpha))$ and $\mathsf{Chan}(\alpha)$ is reachable.
2. $\mathsf{Chan}(\alpha)$ is reachable if there exists a reachable ν such that $\mathsf{Chan}(\alpha) \in \mathsf{refs}_{(\vec{e},h)}(\nu)$.

It is important that reachability is an inductive definition—a coinductive definition would trivially consider all cycles to be reachable.

Definition 1.6.6 (Full reachability ⚙)**.** A configuration $(\vec{e}, h)$ is *fully reachable* if all $\nu \in \mathsf{active}(\vec{e}, h)$ are reachable in $(\vec{e}, h)$.

We show equivalence of partial deadlock/leak freedom and full reachability:

Theorem 1.6.7. ⚙ *A configuration* $(\vec{e}, h)$ *is deadlock/leak free if and only if it is fully reachable.*

For ($\Rightarrow$), we show that none of the objects in a deadlock/leak are reachable, and for ($\Leftarrow$), we show that the set of all non-reachable objects is a deadlock/leak.

Theorem 1.6.8 (Full reachability ⚙)**.** *If* $\mathsf{wf}(\vec{e}, h)$*, then* $(\vec{e}, h)$ *is fully reachable.*

This proof goes by waiting induction and closely resembles the global progress proof in Section 1.3.6. By using the equivalence between full reachability and partial deadlock/leak freedom, we also obtain that a partial deadlock/leak cannot occur, and can re-prove global progress using reachability.

Corollary 1.6.9 (Partial deadlock/leak freedom ⚙)**.** *If* $\mathsf{wf}(\vec{e}, h)$*, then* $(\vec{e}, h)$ *is deadlock/leak free.*

Corollary 1.6.10 (Global progress' ⚙). *If* $\mathsf{wf}(\vec{e}, \mathrm{h})$ *and* $\mathsf{active}(\vec{e}, \mathrm{h}) \neq \emptyset$, *then* $(\vec{e}, \mathrm{h})$ *can step.*

The proof of Theorem 1.6.10 uses Theorem 1.6.8, which gives that active objects are reachable. We then find a thread that can step by straightforward induction on the reachability predicate. Alternatively, we can go via Theorem 1.6.9: if none of the threads can step, then the set of all active threads and channels is a deadlock/leak.

Combined with the proofs that the initial configuration ρ of well-typed program satisfies $\mathsf{wf}(\rho)$, and that $\mathsf{wf}(\rho)$ is preserved by the operational semantics (Section 1.3.5), we obtain partial deadlock/leak freedom, full reachability, and global progress for any well-typed program.

1.7 MECHANIZATION IN COQ

Using the Coq proof assistant (Coq Team, 2021) we have mechanized the generic connectivity graph method and its concrete instantiation to our session-typed language. Our mechanization starts with a library for undirected graphs and their acyclicity described in Jacobs et al. (2021). On top of this, we build a library for connectivity graphs and waiting induction (Section 1.4). We combine connectivity graphs with separation logic (Section 1.3.3) to define the generic well-formedness predicate and the separation logic local transformation lemmas (Section 1.5). We instantiate our library by formalizing the language from Section 1.2 with its extensions from Section 1.6. This involves defining the syntax, type system, and operational semantics. For the language-specific parts of our deadlock and leak freedom proof, we define the run-time type system (Section 1.3.3) and the local well-formedness condition (Section 1.3.4). We then prove preservation using our local transformation rules in separation logic (Section 1.3.5), and progress using our principle of waiting induction (Section 1.3.6). We have also mechanized all the extensions (Section 1.6), including unrestricted types (Section 1.6.1), equi-recursive types (Section 1.6.2), and the theorems about reachability and partial deadlock/leak freedom (Section 1.6.3).

LINE COUNTS The parametric connectivity graph library is 4999 LOC, the language definition is 451 LOC, and the language-specific deadlock and leak freedom proofs are 1688 LOC.

EXTERNAL DEPENDENCIES AND COQ FEATURES THAT WE USE We use the std++ extended standard library for its results on data structures like lists and finite maps (Coq-std++ Team, 2021). We use the Iris Proof Mode for tactics-based separation logic proofs (Krebbers et al., 2017, 2018). To represent recursive types (Section 1.6.2), we use the technique by Gay et al. (2020) based on coinductive types combined with Coq's generalized rewriting mechanism to reason up to the congruence $\equiv$ (Sozeau, 2009).

1.8 RELATED WORK

SESSION TYPES The line of works most closely related to ours are derivatives of Wadler (2012)'s GV, a linear functional language with session types inspired by Gay and Vasconcelos (2010). Whereas Gay and Vasconcelos's calculus does not enjoy the property of deadlock freedom, Wadler's GV and its derivatives (Lindley and Morris, 2015, 2016b, 2017; Fowler et al., 2019, 2021) do. For Wadler's GV, deadlock freedom follows from its translation to CP (Classical Processes) Wadler (2012), for which deadlock freedom holds by cut elimination. Lindley and Morris (2015) then concretize the progress statement by introducing the definition of a deadlocked configuration and proving deadlock freedom using a small-step operational semantics. They also give translations between GV and CP and show that both directions of the translation preserve reductions, unlike previous translations from GV to CP. Subsequently, Lindley and Morris (2015)'s GV has been extended to support least and greatest fixed points (Lindley and Morris, 2016b), exceptions (Fowler et al., 2019), and polymorphism (Lindley and Morris, 2017). A recent extension of GV (Fowler et al., 2021) moreover simplifies GV's meta theory by making process equivalence type preserving. The extension adopts the idea of a hypersequent (Avron, 1991) from (Montesi and Peressotti, 2018; Kokke et al., 2019), yielding Hypersequent GV (HGV).

Like the GV derivatives, our language is a functional language with session-typed channels. Our notion of a connectivity graph moreover bears a resemblance to HGV's abstract process structure (APS), introduced to reason about the acyclic forest structure of a process configuration. However, whereas abstract process structures are defined over hyperenvironments and channel names, our connectivity graph is parametric in its vertices, labels, and edges. More importantly, our connectivity graph is at the core of a proof method for deadlock freedom, fully mechanized in Coq, that uses separation logic and is parametric in its key results. Besides these conceptual differences, there are various technical differences between our formalization and GV formalizations, and even among the different GV variants (such as a synchronous versus an asynchronous semantics). Our formalization uses a standard operational semantics whereas many GV variants use structural congruences and binders for channnel names. In our graph, not only the threads but also channels are vertices, and the edges are directed. Since reasoning about syntax up to equivalence (*e.g.*, structural congruence or α-equivalence) is cumbersome in a proof assistant like Coq, we believe that our operational semantics is better suited for mechanization (and perhaps closer to how these structures are represented on real computers). Orthogonally, we do not tie channel closing to thread termination and allow close everywhere. As a result our language readily accommodates a forest topology without the need for a special connective, such as mix as used by Fowler et al. (2021).

Earlier non-mechanized work has proved deadlock freedom for a π-calculus using a graphical approach (Carbone and Debois, 2010). This is the earliest work that

we are aware of that describes an explicit connection between deadlock freedom and acyclicity of a graph. Their graphical representation is, however, an undirected graph between processes, whereas our connectivity graphs (when instantiated for our language) are directed graphs between threads and channels. Furthermore, their graphs are unlabeled, whereas our connectivity graphs are labeled with session types.

More distantly, our language is related to Toninho et al. (2013); Toninho (2015)'s language SILL, which embeds session-typed processes into a functional core language via a contextual monad. The language is based on the Curry-Howard correspondence established by between intuitionistic linear logic and session-typed π-calculus. Deadlock freedom of SILL follows thus as a consequence. Due to its modal separation, SILL does not allow mixing of functional and session terms freely, in contrast to GV and our language. The seminal paper by Caires and Pfenning (2010) and Toninho (2015)'s thesis spurred a series of derivatives, similarly to Wadler's CP and GV, accommodating, for example, polymorphism (Caires et al., 2013; Pérez et al., 2014), work analysis (Das et al., 2018), and information flow control (Derakhshan et al., 2021). Due to their connection to intuitionistic linear logic, all these works guarantee deadlock freedom. However, unlike ours, none of these deadlock freedom proofs have been mechanized in a proof assistant.

A derivative of SILL, SILL_S (Balzer and Pfenning, 2017), introduces a controlled form of aliasing through a stratification of linear and shared session types with adjoint modalities (Pfenning and Griffith, 2015; Benton, 1994; Reed, 2009b) to support multiple-client scenarios. Whereas the resulting language reclaims the expressiveness of the untyped asynchronous π-calculus for session-typed languages (Balzer et al., 2018), it also sacrifices deadlock freedom (rectified by its successor SILL_S^+ (Balzer et al., 2019)). Recent extensions of classical linear logic session types contribute another approach to softening the rigidity of linear session types to support multiple client sessions and nondeterminism (Qian et al., 2021) and memory cells and nondeterministic updates (Rocha and Caires, 2021), respectively. Whereas neither of these recent approaches reclaim the full expressiveness of unrestricted sharing, they keep the logical foundation intact and thus uphold deadlock freedom. However, none of these works have been mechanized in a proof assistant.

Prior to the development of logic-based session types (Caires and Pfenning, 2010; Wadler, 2012), deadlock freedom in session-typed calculi (Vasconcelos, 2012) was guaranteed only for processes interacting on a *single* session—interleaving of blocking actions on different sessions could easily result in deadlocks. To address limitations of classical binary session types, Honda et al. (2008) introduced *multiparty* session types, where sessions are described by so-called global types that capture the interactions between an arbitrary number of session participants. Given some well-formedness constraints, global types can ensure that a collection of processes correctly implement the global behavior in a deadlock-free way. However, these global type-based approaches do not ensure deadlock freedom in the presence of higher-order channels, interleaved sessions, dynamic channel creation, or dynamic

thread creation. To remedy the deficiency various extensions at increasing degrees of complexity were introduced. For example, Bettini et al. (2008) and Coppo et al. (2016) track usage orders among interleaved multiparty sessions, ruling out cyclic dependencies but also restricting recursion. Our approach instead supports higher-order channels, general recursion, and deadlock freedom solely using a linear type system, by restricting to binary sessions.

SEPARATION LOGIC Separation logic (O'Hearn et al., 2001) is conventionally used in Hoare-style program logics for proving functional correctness, while we use it to define and reason about (run-time) typing judgments. In conventional separation logic, propositions are predicates over heaps (possibly extended with permissions, ghost state, *etc.*), whereas we consider predicates over the outgoing edges of a connectivity graph (which contain types instead of values). The idea of using separation logic to define typing judgments for linear languages has been explored by Rouvoet et al. (2020, 2021) in the context of intrinsically-typed programming in Agda. They present separation-logic based programming abstractions to hide types of references in intrinsically-typed interpreters, and to hide types of labels in intrinsically-typed compilers. As a case study, Rouvoet et al. (2020) use their abstractions to define an intrinsically-typed interpreter for a small session-typed language that guarantees type safety by construction (but not deadlock or resource leak freedom).

Separation logic has also been used to define logical relation models of affine type systems. For example, logical relations in the Iris separation logic (Jung et al., 2015, 2018b) have been used for proving memory safety and data race freedom of Rust (Jung et al., 2018a), as well as type safety of session types (Hinrichsen et al., 2021b). To extend the logical-relations based approach to prove deadlock freedom, a full-fledged separation logic that is capable of proving deadlock freedom is needed. While separation logics and Hoare logics with support for deadlock freedom exist, *e.g.*, (Hamin and Jacobs, 2018; Le et al., 2013; Zhang et al., 2016), they use lock-orders, whose logical expressivity is different from session types. Some separation logics have support for pointed-by assertions (Kassios and Kritikos, 2013), which can be used to reason about memory leak freedom.

Various extensions of separation logic that incorporate session-type based mechanisms to reason about message-passing programs have been developed, *e.g.*, Francalanza et al. (2011); Lozes and Villard (2012); Craciun et al. (2015); Oortwijn et al. (2016); Hinrichsen et al. (2020, 2021a). The goal of these logics is different from ours—they are full-fledged Hoare logics aimed at proving functional correctness instead of deadlock freedom. On the other hand, we use the assertion layer of separation to hide bookkeeping in the definition of run-time typing judgments, and to describe connectivity graph transformations in an abstract and generic way.

MECHANIZED RESULTS OF SESSION TYPES Thiemann (2019) proves type safety of a linear λ-calculus with session types that is inspired by GV. They do not prove

deadlock or memory leak freedom. Their mechanization involves an extensive amount of bookkeeping to keep track of resources. Rouvoet et al. (2020) streamlined this approach via separation logic (see discussion above).

Hinrichsen et al. (2021b) prove type safety for a comprehensive session-typed language with locks, subtyping and polymorphism using Iris in Coq. Their type system is affine, which means that deadlocks are considered safe (their receive operation will spin if the buffer is empty). Their proof is based is on logical relations instead of progress and preservation (see discussion above).

Tassarotti et al. (2017) prove correctness of a compiler for an affine session-typed language using Iris in Coq. The operational semantics of their source language is similar to ours, while channels are compiled to an implementation involving linked lists in the target. Their compiler is proved to be termination preserving, so a target program deadlocks iff the source deadlocks.

More distantly, there also exist various mechanized results involving π-calculus. Goto et al. (2016) prove type safety for a π-calculus with a polymorphic session type system in Coq. Their type system allows dropping channels, and hence does not enjoy deadlock nor memory leak freedom. Ciccone and Padovani (2020) mechanize dependent binary session session types by embedding them into a π-calculus in Agda. They prove subject reduction (*i.e.,* preservation) and that typing is preserved by structural congruence. Neither deadlock freedom nor leak freedom is proved. Castro-Perez et al. (2020) present a framework for locally-nameless representations of π-calculus in Coq. They use their framework to prove subject reduction (*i.e.,* preservation) of a type system for binary session types. Neither deadlock freedom nor leak freedom is proved. Their framework is used by Castro-Perez et al. (2021) to mechanize a DSL for multiparty communication in Coq based on asynchronous multiparty session types. They prove deadlock freedom w.r.t. a global type, but do not prove deadlock freedom in the presence of higher-order channels, interleaved sessions, dynamic channel creation, or dynamic thread creation.

Gay et al. (2020) study various notions of duality in Agda, and show that distribution laws for duality over the recursion operator are unsound. Unlike the other mechanized results discussed so far, they focus on the static instead of dynamic semantics of session types. We have adapted their approach of using coinductive types for mechanizing general recursive session types (see Section 1.7). Keizer et al. (2021) use coalgebras to model session types in a non-mechanized setting.

More distantly related are mechanized versions of cut elimination of linear logic (Reed, 2009a; Chaudhuri et al., 2019), which by Curry-Howard relates to deadlock freedom of intuitionistic session types. The authors were incentivized by mistakes in various existing, non-mechanized proofs. However, whereas a cut elimination proof concerns a logical inference system only, our proof of deadlock freedom encompasses a typed programming language with operational semantics, requiring us to reason not only about its statics but also it execution semantics.

Moreover, our language includes features such as recursive types (Section 1.6.2) that break cut elimination.

Mechanization results, lastly, also exist for choreographic languages (Montesi, 2021). Cruz-Filipe et al. (2021a) mechanize choreography compilation in Coq for the choreographic language Core Choreographies (CC) introduced by Cruz-Filipe et al. (2021b). CC supports recursion and its semantics has been formalized in Coq by Cruz-Filipe et al. (2021b). Key results of the formalization include determinism, confluence, and deadlock-freedom by design as well as Turing completeness.

PROCESS CALCULI The addition of channel usage information to types in a concurrent, message-passing setting was pioneered by Kobayashi (1997); Igarashi and Kobayashi (1997); Kobayashi et al. (1999), who applied the idea to deadlock prevention in the π-calculus and later to more general properties (Igarashi and Kobayashi, 2001, 2004), giving rise to a generic system that can be instantiated to produce a variety of concrete typing disciplines for the π-calculus. Typically, types are augmented with the relative ordering of channel actions, with the type system ensuring that the transitive closure of such orderings forms a strict partial order, ensuring deadlock-freedom. Building on this, Kobayashi (2002) proposed type systems that ensure a stronger property, dubbed lock freedom, and variants that are amenable to type inference (Kobayashi et al., 2000). Kobayashi (2006) extended this to account for recursive processes and type inference. Kobayashi-style systems have also been adopted for session-typed languages (Dardha and Gay, 2018; Balzer et al., 2019).

1.9 FUTURE WORK

We have used our connectivity graph method to give a mechanized proof of deadlock and memory-leak freedom for binary session types. Since connectivity graphs are not restricted to two incoming edges per channel, we would like to explore language designs with a version of multiparty session types that supports dynamic thread and channel creation, and higher order channels, but still enjoys global progress from typing in a manner similar to binary session types (*i.e.*, without additional mechanisms such as orders or priorities). Second, we would like to explore whether other concurrency mechanisms such as locks and barriers could be handled by our method.

ACKNOWLEDGMENTS We thank the anonymous reviewers for their helpful feedback, and especially for the criticism that global progress is a very weak property if one thread is in an infinite loop, which is what led us to develop Section 1.6.3. We are grateful to Herman Geuvers, Fabrizio Montesi, Ike Mulder, Arjen Rouvoet, Bernardo Toninho, and Jorge Pérez for discussions about this paper and related work. The second author (Stephanie Balzer) was supported by National Science Foundation

Award No. CCF-1718267. The third author (Robbert Krebbers) was supported by the Dutch Research Council (NWO), project 016.Veni.192.259.

Bibliography

Arnon Avron. 1991. Hypersequents, Logical Consequence and Intermediate Logics for Concurrency. *Annals of Mathematics and Artificial Intelligence* (1991). https://doi.org/10.1007/BF01531058

Stephanie Balzer and Frank Pfenning. 2017. Manifest Sharing with Session Types. ICFP (2017). https://doi.org/10.1145/3110281

Stephanie Balzer, Frank Pfenning, and Bernardo Toninho. 2018. A Universal Session Type for Untyped Asynchronous Communication. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2018.30

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *ESOP*. https://doi.org/10.1007/978-3-030-17184-1_22

Nick Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *CSL*. https://doi.org/10.1007/BFb0022251

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *CONCUR*. https://doi.org/10.1007/978-3-540-85361-9_33

Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. 2013. Behavioral Polymorphism and Parametricity in Session-Based Communication. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_19

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR*. https://doi.org/10.1007/978-3-642-15375-4_16

Marco Carbone and Søren Debois. 2010. A Graphical Approach to Progress for Structured Communication in Web Services. In *ICE*. https://doi.org/10.4204/EPTCS.38.4

David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, and Nobuko Yoshida. 2021. Zooid: A DSL for Certified Multiparty Computation: From Mechanised Metatheory to Certified Multiparty Processes. In *PLDI*. https://doi.org/10.1145/3453483.3454041

David Castro-Perez, Francisco Ferreira, and Nobuko Yoshida. 2020. EMTST: Engineering the Meta-theory of Session Types. In *TACAS*. https://doi.org/10.1007/978-3-030-45237-7_17

Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. 2019. Formalized Meta-Theory of Sequent Calculi for Linear Logics. *Theoretical Computer Science* (2019). https://doi.org/10.1016/j.tcs.2019.02.023

Ruofei Chen and Stephanie Balzer. 2020. Ferrite: A Judgmental Embedding of Session Types in Rust. *CoRR* (2020). arXiv:2009.13619 https://arxiv.org/abs/2009.13619

Luca Ciccone and Luca Padovani. 2020. A Dependently Typed Linear π-Calculus in Agda. In *PPDP*. https://doi.org/10.1145/3414080.3414109

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *MSCS* (2016). https://doi.org/10.1017/S0960129514000188

The Coq-std++ Team. 2021. An extended "standard library" for Coq. Available online at https://gitlab.mpi-sws.org/iris/stdpp.

The Coq Team. 2021. The Coq Proof Assistant. https://doi.org/10.5281/zenodo.4501022

Florin Craciun, Tibor Kiss, and Andreea Costea. 2015. Towards a Session Logic for Communication Protocols. In *ICECCS*. https://doi.org/10.1109/ICECCS.2015.33

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021a. Certifying Choreography Compilation. In *ICTAC*. https://doi.org/10.1007/978-3-030-85315-0_8

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021b. Formalising a Turing-Complete Choreographic Language in Coq. In *ITP*. https://doi.org/10.4230/LIPIcs.ITP.2021.15

Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *FOSSACS*. https://doi.org/10.1007/978-3-319-89366-2_5

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018. Work Analysis with Resource-Aware Session Types. In *LICS*. https://doi.org/10.1145/3209108.3209146

Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. 2021. Session Logical Relations for Noninterference. In *LICS*. https://doi.org/10.1109/LICS52264.2021.9470654

Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. 2006. Session Types for Object-Oriented Languages. In *ESOP*. https://doi.org/10.1007/11785477_20

Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. 2021. Separating Sessions Smoothly. In *CONCUR*. https://doi.org/10.4230/LIPIcs.CONCUR.2021.36

Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. 2019. Exceptional Asynchronous Session Types: Session Types Without Tiers. POPL (2019). https://doi.org/10.1145/3290341

Adrian Francalanza, Julian Rathke, and Vladimiro Sassone. 2011. Permission-Based Separation Logic for Message-Passing Concurrency. *LMCS* (2011). https://doi.org/10.2168/LMCS-7(3:7)2011

Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. 2020. Duality of Session Types: The Final Cut. In *PLACES*. https://doi.org/10.4204/EPTCS.314.3

Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *JFP* (2010). https://doi.org/10.1017/S0956796809990268

Matthew A. Goto, Radha Jagadeesan, Alan Jeffrey, Corin Pitcher, and James Riely. 2016. An Extensible Approach to Session Polymorphism. *MSCS* (2016). https://doi.org/10.1017/S0960129514000231

Jafar Hamin and Bart Jacobs. 2018. Deadlock-Free Monitors. In *ESOP*. https://doi.org/10.1007/978-3-319-89884-1_15

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). https://doi.org/10.5555/3002812

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2020. Actris: Session-Type Based Reasoning in Separation Logic. POPL (2020). https://doi.org/10.1145/3371074

Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. 2021a. Actris 2.0: Asynchronous Session-Type Based Reasoning in Separation Logic. (2021). https://arxiv.org/abs/2010.15030v1 Manuscript.

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *CONCUR*. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP*. https://doi.org/10.1007/BFb0053567

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *POPL*. https://doi.org/10.1145/1328438.1328472

Atsushi Igarashi and Naoki Kobayashi. 1997. Type-Based Analysis of Communication for Concurrent Programming Languages. In *SAS*. https://doi.org/10.1007/BFb0032742

Atsushi Igarashi and Naoki Kobayashi. 2001. A Generic Type System for the Pi-calculus. In *POPL*. https://doi.org/10.1145/360204.360215

Atsushi Igarashi and Naoki Kobayashi. 2004. A Generic Type System for the Pi-calculus. *Theoretical Computer Science* (2004). https://doi.org/10.1016/S0304-3975(03)00325-6

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017. Gradual session types. ICFP (2017). https://doi.org/10.1145/3110282

Keigo Imai, Nobuko Yoshida, and Shoji Yuen. 2019. Session-Ocaml: A Session-Based Library with Polarities and Lenses. *Science of Computer Programming* (2019). https://doi.org/10.1016/j.scico.2018.08.005

Keigo Imai, Shoji Yuen, and Kiyoshi Agusa. 2010. Session Type Inference in Haskell. In *PLACES*. https://doi.org/10.4204/EPTCS.69.6

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2021. Appendix and Coq mechanization of "Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic". The most recent version is at https://github.com/julesjacobs/cgraphs..

Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session Types for Rust. In *WGP*. https://doi.org/10.1145/2808098.2808100

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the Foundations of the Rust Programming Language. POPL (2018). https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* (2018). https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *POPL*. https://doi.org/10.1145/2676726.2676980

Ioannis T. Kassios and Eleftherios Kritikos. 2013. A Discipline for Program Verification Based on Backpointers and Its Use in Observational Disjointness. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_10

Alex C. Keizer, Henning Basold, and Jorge A. Pérez. 2021. Session Coalgebras: A Coalgebraic View on Session Types and Communication Protocols. https://doi.org/10.1007/978-3-030-72019-3_14

Naoki Kobayashi. 1997. A Partially Deadlock-Free Typed Process Calculus. In *LICS*. https://doi.org/10.1109/LICS.1997.614941

Naoki Kobayashi. 2002. A Type System for Lock-Free Processes. *I&C* (2002). https://doi.org/10.1006/inco.2002.3171

Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR*. https://doi.org/10.1007/11817949_16

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *TOPLAS* (1999). https://doi.org/10.1145/330249.330251

Naoki Kobayashi, Shin Saito, and Eijiro Sumii. 2000. An Implicitly-Typed Deadlock-Free Process Calculus. In *CONCUR*. https://doi.org/10.1007/3-540-44618-4_35

Wen Kokke. 2019. Rusty Variation: Deadlock-free Sessions with Failure in Rust. In *ICE*. https://doi.org/10.4204/EPTCS.304.4

Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better Late Than Never: a Fully-Abstract Semantics for Classical Processes. POPL (2019). https://doi.org/10.1145/3290337

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. ICFP (2018). https://doi.org/10.1145/3236772

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *POPL*. https://doi.org/10.1145/3009837.3009855

Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo. 2013. An Expressive Framework for Verifying Deadlock Freedom. In *ATVA*. https://doi.org/10.1007/978-3-319-02444-8_21

Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *ESOP*. https://doi.org/10.1007/978-3-662-46669-8_23

Sam Lindley and J. Garrett Morris. 2016a. Embedding Session Types in Haskell. In *Haskell Symposium*. https://doi.org/10.1145/2976002.2976018

Sam Lindley and J. Garrett Morris. 2016b. Talking Bananas: Structural Recursion For Session Types. In *ICFP*. https://doi.org/10.1145/2951913.2951921

Sam Lindley and J. Garrett Morris. 2017. Lightweight Functional Session Types. In *Behavioural Types: from Theory to Tools*.

Étienne Lozes and Jules Villard. 2012. Shared Contract-Obedient Endpoints. In *ICE*. https://doi.org/10.4204/EPTCS.104.3

Fabrizio Montesi. 2021. Introduction to Choreographies. (2021). Accepted for publication by Cambridge University Press.

Fabrizio Montesi and Marco Peressotti. 2018. Classical Transitions. *CoRR* (2018). arXiv:1803.01049 http://arxiv.org/abs/1803.01049

Peter W. O'Hearn and David J. Pym. 1999. The Logic Of Bunched Implications. *Bulletin of Symbolic Logic* (1999). https://doi.org/10.2307/421090

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL*. https://doi.org/10.1007/3-540-44802-0_1

Wytse Oortwijn, Stefan Blom, and Marieke Huisman. 2016. Future-based Static Analysis of Message Passing Programs. In *PLACES*. https://doi.org/10.4204/EPTCS.211.7

Luca Padovani. 2017. A simple library implementation of binary sessions. *JFP* (2017). https://doi.org/10.1017/S0956796816000289

Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2014. Linear Logical Relations and Observational Equivalences for Session-Based Concurrency. *I&C* (2014). https://doi.org/10.1016/j.ic.2014.08.001

Frank Pfenning and Dennis Griffith. 2015. Polarized Substructural Session Types. In *FoSSaCS*. https://doi.org/10.1007/978-3-662-46678-0_1

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). https://doi.org/10.5555/509043

Riccardo Pucella and Jesse A. Tov. 2008. Haskell session types with (almost) no class. In *Haskell Symposium*. https://doi.org/10.1145/1411286.1411290

Zesen Qian, G. A. Kavvos, and Lars Birkedal. 2021. Client-Server Sessions in Linear Logic. ICFP (2021). https://doi.org/10.1145/3473567

Jason Reed. 2009a. *A Hybrid Logical Framework*. Ph. D. Dissertation. Carnegie Mellon University.

Jason Reed. 2009b. A Judgmental Deconstruction of Modal Logic. (2009). http://www.cs.cmu.edu/~jcreed/papers/jdml.pdf Unpublished manuscript.

Pedro Rocha and Luís Caires. 2021. *Propositions-as-Types and Shared State*. Technical Report. NOVA LINCS. https://doi.org/10.1145/3473584

Arjen Rouvoet, Robbert Krebbers, and Eelco Visser. 2021. Intrinsically Typed Compilation With Nameless Labels. POPL (2021). https://doi.org/10.1145/3434303

Arjen Rouvoet, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. 2020. Intrinsically-Typed Definitional Interpreters for Linear, Session-Typed Languages. In *CPP*. https://doi.org/10.1145/3372885.3373818

Alceste Scalas and Nobuko Yoshida. 2016. Lightweight Session Programming in Scala. In *ECOOP*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.21

Matthieu Sozeau. 2009. A New Look at Generalized Rewriting in Type Theory. *JFR* (2009). https://doi.org/10.6092/issn.1972-5787/1574

Joseph Tassarotti, Ralf Jung, and Robert Harper. 2017. A Higher-Order Logic for Concurrent Termination-Preserving Refinement. In *ESOP*. https://doi.org/10.1007/978-3-662-54434-1_34

Peter Thiemann. 2019. Intrinsically-Typed Mechanized Semantics for Session Types. In *PPDP*. https://doi.org/10.1145/3354166.3354184

Bernardo Toninho. 2015. *A Logical Foundation for Session-Based Concurrent Computation*. Ph. D. Dissertation. Carnegie Mellon University and New University of Lisbon.

Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP*. https://doi.org/10.1007/978-3-642-37036-6_20

Vasco T. Vasconcelos. 2012. Fundamentals of Session Types. *I&C* (2012). https://doi.org/10.1016/j.ic.2012.05.002

Philip Wadler. 2012. Propositions as Sessions. In *ICFP*. https://doi.org/10.1145/2364527.2364568

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *I&C* (1994). https://doi.org/10.1006/inco.1994.1093

Dan Zhang, Dragan Bosnacki, Mark van den Brand, Cornelis Huizing, Bart Jacobs, Ruurd Kuiper, and Anton Wijs. 2016. Verifying Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations. In *MODELSWARD*. https://doi.org/10.1007/978-3-319-66302-9_13