A Self-Dual Distillation of Session Types (Pearl)

Jules Jacobs ✉
Radboud University Nijmegen

Abstract

We introduce $\bar{\lambda}$ ("lambda-barrier"), a minimal extension of linear λ-calculus with concurrent communication, which adds only a *single* new **fork** construct for spawning threads. It is inspired by GV, a session-typed functional language also based on linear λ-calculus. Unlike GV, $\bar{\lambda}$ strives to be as simple as possible, and adds no new operations other than **fork**, no new type formers, and no explicit definition of session type duality. Instead, we use linear function function type $\tau_1 \multimap \tau_2$ for communication between threads, which is dual to $\tau_2 \multimap \tau_1$, *i.e.,* the function type constructor is dual to itself. Nevertheless, we can encode session types as $\bar{\lambda}$ types, GV's channel operations as $\bar{\lambda}$ terms, and show that this encoding is type-preserving. The linear type system of $\bar{\lambda}$ ensures that all programs are deadlock-free and satisfy global progress, which we prove in Coq. Because of $\bar{\lambda}$'s minimality, these proofs are simpler than mechanized proofs of deadlock freedom for GV.

2012 ACM Subject Classification Software and its engineering → Concurrent programming languages

Keywords and phrases Linear types, concurrency, lambda calculus, session types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.23

Supplementary Material *Software (Mechanized proofs)*: `https://zenodo.org/record/6560443`

1 Introduction

Session types [16, 15] are types for communication channels, that can be used to verify that programs follow the communication protocol specified by a channel's session type. Gay and Vasconcelos [13] embed session types in a linear λ-calculus. Whereas Gay and Vasconcelos' calculus [13] did not yet ensure deadlock freedom, Wadler's subsequent GV [31] and its derivatives [21, 23, 24, 11, 10] guarantee that all well-typed programs are deadlock free.

In order to add session types to linear λ-calculus, one adds (linear) session type formers for typing channel protocols and their corresponding operations: $\mathbf{!}\tau.s$ (send a message of type τ, continue with protocol s), $\mathbf{?}\tau.s$ (receive a message of type τ, continue with protocol s), $s_1 \oplus s_2$ (send choice between protocols s_1 and s_2), $s_1 \,\&\, s_2$ (receive choice between protocols s_1 and s_2), and End (close channel). One also adds a **fork** operation for creating a thread and a pair of dual channels. For this, we need a definition of duality, with **!** dual to **?**, $\oplus$ dual to &, and End dual to itself.

There have been efforts for simpler systems, such as an encoding of session types into ordinary π-calculus types [19, 7, 8], and *minimal session types* [2], which decompose multi-step session types into single-step session types in a π-calculus. Single-shot synchronization primitives have also been used in the implementation of a session-typed channel libraries [30, 27, 20].

We show that linear λ-calculus is also an excellent substrate on which to build a minimal concurrent calculus with communication, and introduce $\bar{\lambda}$ ("lambda-barrier"), which adds only a *single* new **fork** construct for spawning threads. It is inspired by GV, a session-typed functional language that is also based on linear λ-calculus. Unlike GV, $\bar{\lambda}$ strives to be as simple as possible, and adds no new operations other than **fork**, no new type formers, and no explicit definition of duality. Instead, we use the linear function type $\tau_1 \multimap \tau_2$ for communication between threads, which is dual to $\tau_2 \multimap \tau_1$, *i.e.,* the function type constructor is dual to itself. Nevertheless, we can encode session types as $\bar{\lambda}$ types, GV's channel operations

© Jules Jacobs;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 23; pp. 23:1–23:22

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

as $\overline{\lambda}$ terms, and show that this encoding is type-preserving. A key difference with CPS encodings of GV [22, 23], which are whole-program, is that our encoding is local, and uses $\overline{\lambda}$'s built-in concurrency.

Like GV, all well-typed $\overline{\lambda}$ programs are automatically *deadlock free*, and therefore satisfy *global progress*. We prove this property in Coq. Because of $\overline{\lambda}$'s minimality, these proofs are simpler and shorter than mechanized proofs of deadlock freedom for GV.

The rest of this article is structured as follows:

- An introduction to $\overline{\lambda}$ by example (Section 2).
- The $\overline{\lambda}$ type system and operational semantics (Section 3).
- Encoding session types in $\overline{\lambda}$ (Section 4).
- How to prove global progress and deadlock freedom for $\overline{\lambda}$ (Section 5).
- Extending $\overline{\lambda}$ with unrestricted and recursive types (Section 6).
- Mechanizing the meta-theory of $\overline{\lambda}$ in Coq (Section 7).
- Related work (Section 8).
- Concluding remarks (Section 9).

2 The $\overline{\lambda}$ language by example

The $\overline{\lambda}$ language consists of linear λ-calculus with a single extension: **fork**.[1] Let us look at an example:

$$\textbf{let } x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1)) \textbf{ in print}(1 + x\ 0)$$

This program forks off a new thread, which also creates *communication barriers* x and x' to communicate between the threads. The barrier x gets returned to the main thread, and x' gets passed to the child thread. These barriers are functions, and a call to a barrier will block until the other side is also trying to synchronize, and will then atomically exchange the values passed as an argument. The example runs as follows:

- When $x'\ 1$ is called, it will block until $x\ 0$ is also called, and vice versa.
- The call $x'\ 1$ will then return 0, and the call $x\ 0$ will return 1.

Thus, the program will print 0 2 or 2 0, depending on which thread prints first. In $\overline{\lambda}$, these barriers are *linear*, so they must be used exactly once:

$$\textbf{fork}(\lambda x.\ \textbf{print}(1)) \quad \text{Error! Must use x.}$$
$$\textbf{fork}(\lambda x.\ \textbf{print}(x\ 0 + x\ 1)) \quad \text{Error! Can't use x twice.}$$

The type of **fork** is:

$$\textbf{fork} : ((\tau_1 \multimap \tau_2) \multimap \mathbf{1}) \to (\tau_2 \multimap \tau_1)$$

where $\multimap$ is the type of linear functions. Linearity allows us to encode *session types* in $\overline{\lambda}$ (Section 4), and ensures that all well-typed $\overline{\lambda}$ programs are *deadlock-free* (Section 5), which would not be the case without linearity. Nevertheless, linearity may seem like a critical limitation: can a child thread communicate with its parent thread only *once*?! Luckily, two features of λ-calculus, namely the ability for closures to capture values from their lexical environment, and the ability to pass functions as arguments to other functions, means that

[1] For the examples we also use **print**, to be able to talk about the operational behavior of programs.

the restriction is not as severe as it may seem. Let's look at an example that uses those two features:[2]

$$\textbf{let } x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1))$$
$$\textbf{let } y = \textbf{fork}(\lambda y'.\ y'\ x)$$
$$\textbf{print}(1 + y\ ()\ 0)$$

We fork off a new thread (line 1) and store its barrier in x. We then fork off another thread (line 2), and pass the barrier x into the λ-expression of the new thread. We call y (), which returns x, because the thread of y calls $y'\ x$. Finally, we pass 0 into the returned x, so this example behaves the same as the first example: it prints 0 2 or 2 0.

We can use the ability to capture barriers in the λ-expression of a fork, and the ability to send barriers over barriers, to set up long-running communication between two threads:

$$\begin{aligned}&\textbf{let } x_1 = \textbf{fork}(\lambda x_1'.\ \textbf{let } (x_2', n_1) = x_1'\ ()\\ &\qquad\qquad\qquad\quad \textbf{let } (x_3', n_2) = x_2'\ ()\ \textbf{in } x_3'\ (n_1 + n_2))\\ &\textbf{let } x_2 = \textbf{fork}(\lambda x_2'.\ x_1\ (x_2', 1))\\ &\textbf{let } x_3 = \textbf{fork}(\lambda x_3'.\ x_2\ (x_3', 2))\\ &\textbf{print}(x_3\ ())\end{aligned}$$

Let us focus on the body of the first fork. The forked thread firstly synchronizes with its barrier, via x_1' (). This call will return a pair (x_2', n_1) of a *new* barrier x_2', and a number n_1. It then synchronizes with the new barrier, via x_2' (), which returns *another* pair (x_3', n_2), giving yet another barrier x_3' and another number n_2. In the last step, it sends the number $n_1 + n_2$ back to the main thread, via $x_3'\ (n_1 + n_2)$.

Let us now focus on how the main thread arranges this sequence of communications. The main thread first forks off a *messenger thread* $\textbf{fork}(\lambda x_2'.\ x_1\ (x_2', 1))$. The purpose of this thread is to send the message $(x_2', 1)$ over x_1, where x_2' is the barrier associated with the messenger thread. The other side of that barrier, x_2, is given to the main thread. The main thread now forks off another messenger thread, this time using that new barrier, x_2. This gives the main thread yet another barrier, x_3, from which it receives the final answer, $1 + 2$, via x_3 ().

Note that, like in the asynchronous π-calculus, sending a message involves forking off a tiny thread. Thus, like the asynchronous π-calculus, $\bar{\lambda}$ should be viewed as a theoretical core calculus, and not as a practical way to implement message passing.[3] We can encapsulate this messenger thread pattern in a small library of channel operations:

$$\textbf{send}(c, x) \triangleq \textbf{fork}(\lambda c'.\ c\ (c', x))$$
$$\textbf{receive}(c) \triangleq c\ ()$$
$$\textbf{close}(c) \triangleq c\ ()$$

[2] We omit the **in** keyword if a newline follows **let**, like some functional languages (*e.g.,* F#).

[3] Because the messenger threads are always of a specific form, it might be possible to implement a compiler that recognizes such patterns and implements them more efficiently. After all, the messenger threads do nothing but immediately synchronize with another barrier.

Using this channel library, we can implement the preceding example in the following way:

$$\begin{aligned}
&\textbf{let } x_1 = \textbf{fork}(\lambda x_1'.\ \begin{aligned}[t]&\textbf{let } (x_2', n_1) = \textbf{receive}(x_1')\\ &\textbf{let } (x_3', n_2) = \textbf{receive}(x_2')\\ &\textbf{let } x_4' = \textbf{send}(x_3', n_1 + n_2)\\ &\textbf{close}(x_4'))\end{aligned}\\
&\textbf{let } x_2 = \textbf{send}(x_1, 1)\\
&\textbf{let } x_3 = \textbf{send}(x_2, 2)\\
&\textbf{let } (x_4, n) = \textbf{receive}(x_3)\\
&\textbf{print}(n)\\
&\textbf{close}(x_4)
\end{aligned}$$

Session-typed channels usually also have *choice*, which allows choosing between two continuation protocols. This can be encoded in $\bar{\lambda}$ using sums $\textbf{in}_\textbf{L}(x)$ and $\textbf{in}_\textbf{R}(x)$:

$$\begin{aligned}
\textbf{tell}_\textbf{L}(c) &\triangleq \textbf{fork}(\lambda c'.\ c\ \textbf{in}_\textbf{L}(c'))\\
\textbf{tell}_\textbf{R}(c) &\triangleq \textbf{fork}(\lambda c'.\ c\ \textbf{in}_\textbf{R}(c'))\\
\textbf{ask}(c) &\triangleq c\ ()
\end{aligned}$$

With these operations, we can implement the calculator example of Lindley and Morris [24]. This example allows the client to choose whether they want to add two numbers or negate a number. If the client chooses to add two numbers, they then send two numbers as separate messages, and retrieve the answer using receive. If the client chooses to negate a number, they then send only a single number, and retrieve the answer using receive. This example illustrates the choice between two different protocols for the remaining interaction:

$$\begin{aligned}
&\textbf{let } \mathsf{calc}\ c =\\
&\quad \textbf{match ask}(c) \textbf{ with}\\
&\quad \mid \textbf{in}_\textbf{L}(c) \Rightarrow \begin{aligned}[t]&\textbf{let } (c, n) = \textbf{receive}(c)\\ &\textbf{let } (c, m) = \textbf{receive}(c)\\ &\textbf{close}(\textbf{send}(c, n + m))\end{aligned}\\
&\quad \mid \textbf{in}_\textbf{R}(c) \Rightarrow \begin{aligned}[t]&\textbf{let } (c, n) = \textbf{receive}(c)\\ &\textbf{close}(\textbf{send}(c, -n))\end{aligned}\\
&\quad \textbf{end}
\end{aligned}$$

Extending $\bar{\lambda}$ with recursion (Section 6) allows us to implement unbounded protocols, as illustrated by the following example:

$$\begin{aligned}
&\textbf{let rec } \mathsf{countdown}\ c =\\
&\quad \textbf{match ask}(c) \textbf{ with}\\
&\quad \mid \textbf{in}_\textbf{L}(c) \Rightarrow \textbf{close}(c)\\
&\quad \mid \textbf{in}_\textbf{R}(c) \Rightarrow \begin{aligned}[t]&\textbf{let } (c, n) = \textbf{receive}(c)\\ &\textbf{print}(n)\\ &\textbf{if } n = 0\\ &\ \textbf{then close}(\textbf{tell}_\textbf{L}(c))\\ &\ \textbf{else } \mathsf{countdown}\ (\textbf{send}(\textbf{tell}_\textbf{R}(c), n - 1))\end{aligned}\\
&\quad \textbf{end}
\end{aligned}$$

Given a channel c, the $\mathsf{countdown}\ c$ program first uses $\textbf{ask}(c)$ to ask c if it wants to terminate, and closes the channel if so. Otherwise it receives a number n from c and prints

it. Then it checks if the number $n = 0$, and if so tells the other side to close (using $\mathbf{tell_L}$), and then closes our side. Otherwise it tells the other side that it wants to continue (using $\mathbf{tell_R}$) and sends $n - 1$ to the other side. We can therefore let countdown interact with a copy of itself, provided we start off one of the copies with an initial message:

$\mathsf{countdown}\ \mathbf{send}(\mathbf{tell_R}(\mathbf{fork}(\mathsf{countdown})), 10)$

This program will print the numbers $10\ 9\ 8\ \cdots\ 1\ 0$, in that order. The odd numbers are printed by the main thread, and the even numbers are printed by the child thread.

As we shall see later, this is all type-safe. If we had not started off one of the countdowns with an initial message, and had instead simply done $\mathsf{countdown}\ \mathbf{fork}(\mathsf{countdown})$, then we would have had a static type error.

This way, the ƛ type system ensures that channel protocols are correctly followed, even though the ƛ type system has no session types and no notion of duality and instead simply uses function types $\tau_1 \multimap \tau_2$ for barriers. We do not need an explicit notion of duality because the function type constructor is *self-dual*, in the sense that if $x : \tau_1 \multimap \tau_2$ is a barrier, then the dual barrier with which x will synchronize has type $x' : \tau_2 \multimap \tau_1$. We will see more about encoding session types in ƛ in Section 4.

3 The ƛ type system and operational semantics

Like GV [31] and its derivatives [21, 23, 24, 11, 10], the basis of ƛ is a linear simply typed λ-calculus. We have sums, products, and the linear function type $\tau_1 \multimap \tau_2$. Variables of linear type must be used exactly once: they cannot be duplicated (contracted) or discarded (weakened), so that one must use one of the elimination rules of the type.

$$\tau \in \mathit{Type} ::= \mathbf{0} \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \tau \multimap \tau$$

Our basis linear λ-calculus has the following grammar of expressions, which consists of introduction and elimination forms for each type:

$$\begin{aligned} e \in \mathit{Expr} ::= {}& x \mid () \mid (e, e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\ e \mid e\ e \mid \mathbf{let}\ (x_1, x_2) = e\ \mathbf{in}\ e \mid \\ & \mathbf{match}\ e\ \mathbf{with}\ \ \mathbf{end} \mid \mathbf{match}\ e\ \mathbf{with}\ \mathbf{in_L}(x_1) \Rightarrow e_1;\ \mathbf{in_R}(x_2) \Rightarrow e_2\ \mathbf{end} \end{aligned}$$

The typing rules are standard and can be found in Figure 1.

We now have a substrate to which we will add concurrency constructs. GV [31, 21, 23, 24, 11, 10] introduces concurrency by means of a construct to spawn a new thread, with which we can communicate using a channel. Communication is governed by session types, such that the two endpoints of a channel are typed with dual session types. Instead, ƛ extends the substrate with concurrency in a minimal way, adding one new construct to create new threads:

$$e \in \mathit{Expr} ::= \cdots \mid \mathbf{fork}(e)$$

This is the typing rule for **fork**:

$$\frac{\Gamma \vdash e : (\tau_2 \multimap \tau_1) \multimap \mathbf{1}}{\Gamma \vdash \mathbf{fork}(e) : \tau_1 \multimap \tau_2}$$

Or, as a type signature:

$$\mathbf{fork} : ((\tau_1 \multimap \tau_2) \multimap \mathbf{1}) \to (\tau_2 \multimap \tau_1)$$

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

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \times \tau_2 \qquad \Gamma_2, x_1{:}\tau_1, x_2{:}\tau_2 \vdash e_2 : \tau_3}{\Gamma_1, \Gamma_2 \vdash \textbf{let } (x_1, x_2) = e_1 \textbf{ in } e_2 : \tau_3}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{in}_\textbf{L}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{in}_\textbf{R}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma_1 \vdash e : \tau_1 + \tau_2 \qquad \Gamma_2, x_1{:}\tau_1 \vdash e_1 : \tau' \qquad \Gamma_2, x_2{:}\tau_2 \vdash e_2 : \tau'}{\Gamma_1, \Gamma_2 \vdash \textbf{match } e \textbf{ with in}_\textbf{L}(x_1) \Rightarrow e_1;\ \textbf{in}_\textbf{R}(x_2) \Rightarrow e_2 \textbf{ end} : \tau'}$$

Figure 1 Linear λ-calculus with sums and products (rules for **0** and **1** omitted).

The type of **fork** uses the linear function type. We do not need an explicit notion of duality, like session types do, because the function type constructor is *self-dual*, in the sense that if $x : \tau_1 \multimap \tau_2$ is a barrier, then the dual barrier x' with which x will synchronize has type $x' : \tau_2 \multimap \tau_1$.

3.1 Operational semantics

We use a small-step operational semantics with evaluation contexts. In order to represent barriers, we add barrier literals $\langle k \rangle, k \in \mathbb{N}$ to the expressions. A barrier literal cannot appear in the source program, as the static type system has no typing rule for it. Barrier literals only appear in expressions at runtime when the operational semantics executes a **fork**-step. This gives us the following set of values for the language:

$$v \in \mathit{Val} ::= () \mid (v, v) \mid \textbf{in}_\textbf{L}(v) \mid \textbf{in}_\textbf{R}(v) \mid \lambda x.\ e \mid \langle k \rangle$$

We have four pure head-reduction rules $e \rightsquigarrow_{\text{pure}} e'$, one for λ, one for pairs, and two for sums, as stated in Figure 2. We use evaluation contexts to avoid introducing many congruence rules:[4]

$$K ::= \square \mid (K, e) \mid (v, K) \mid \textbf{in}_\textbf{L}(K) \mid \textbf{in}_\textbf{R}(K) \mid K\ e \mid v\ K \mid \textbf{fork}(K) \mid \textbf{let } (x_1, x_2) = K \textbf{ in } e$$
$$\mid \textbf{match } K \textbf{ with } \ \textbf{end} \mid \textbf{match } K \textbf{ with in}_\textbf{L}(x_1) \Rightarrow e_1;\ \textbf{in}_\textbf{R}(x_2) \Rightarrow e_2 \textbf{ end}$$

We represent a configuration with multiple threads and barriers as a finite map:

$$\rho \in \mathit{Cfg} \triangleq \mathbb{N} \xrightharpoonup{\text{fin}} \mathsf{Thread}(\mathit{Expr}) + \mathsf{Barrier}$$

We define a **configuration step relation** $\rho \overset{i}{\rightsquigarrow} \rho'$. The label $i \in \mathbb{N}$ is used to keep track of which thread or barrier in the configuration takes the step. This has no effect on the operational semantics, but we will later use it to formulate deadlock freedom. The rules for the configuration step relation are given in Figure 2. We have the following five rules, in the order of the figure:

[4] This set of evaluation contexts results in left-to-right evaluation order, but the mechanization has been set up so that the proof scripts work for right-to-left and nondeterministic order as well.

$$(\lambda x.\ e)\ v \rightsquigarrow_{\text{pure}} e[v/x]$$
$$\textbf{let}\ (x_1, x_2) = (v_1, v_2)\ \textbf{in}\ e \rightsquigarrow_{\text{pure}} e[v_1/x_1][v_2/x_2]$$
$$\textbf{match}\ \textbf{in}_\mathbf{L}(v)\ \textbf{with}\ \textbf{in}_\mathbf{L}(x_1) \Rightarrow e_1 \mid \textbf{in}_\mathbf{R}(x_2) \Rightarrow e_2\ \textbf{end} \rightsquigarrow_{\text{pure}} e_1[v/x_1]$$
$$\textbf{match}\ \textbf{in}_\mathbf{R}(v)\ \textbf{with}\ \textbf{in}_\mathbf{L}(x_1) \Rightarrow e_1 \mid \textbf{in}_\mathbf{R}(x_2) \Rightarrow e_2\ \textbf{end} \rightsquigarrow_{\text{pure}} e_2[v/x_2]$$

$$\{n \mapsto \mathsf{Thread}(K[\,e_1\,])\} \overset{n}{\rightsquigarrow} \{n \mapsto \mathsf{Thread}(K[\,e_2\,])\} \quad \text{if } e_1 \rightsquigarrow_{\text{pure}} e_2 \qquad \text{(pure)}$$

$$\{n \mapsto \mathsf{Thread}(K[\,\textbf{fork}(v)\,])\} \overset{n}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K[\ \langle k\rangle\ ]) \\ k \mapsto \mathsf{Barrier} \\ m \mapsto \mathsf{Thread}(v\ \ \langle k\rangle) \end{Bmatrix} \qquad \text{(fork)}$$

$$\begin{Bmatrix} n \mapsto \mathsf{Thread}(K_1[\ \langle k\rangle\ \ v_1\,]) \\ k \mapsto \mathsf{Barrier} \\ m \mapsto \mathsf{Thread}(K_2[\ \langle k\rangle\ \ v_2\,]) \end{Bmatrix} \overset{k}{\rightsquigarrow} \begin{Bmatrix} n \mapsto \mathsf{Thread}(K_1[\,v_2\,]) \\ m \mapsto \mathsf{Thread}(K_2[\,v_1\,]) \end{Bmatrix} \qquad \text{(sync)}$$

$$\{n \mapsto \mathsf{Thread}(())\} \overset{n}{\rightsquigarrow} \{\} \qquad \text{(exit)}$$

$$\rho_1 \uplus \rho' \overset{i}{\rightsquigarrow} \rho_2 \uplus \rho' \quad \text{if } \rho_1 \overset{i}{\rightsquigarrow} \rho_2 \quad (\uplus \text{ is disjoint union}) \qquad \text{(frame)}$$

Figure 2 The operational semantics of ƛ.

(pure) A rule for pure reductions for a single thread.
(fork) A rule for forking a new thread, which adds the new thread and a barrier k to the configuration. The two threads get access to the barrier via the barrier literal $\langle k\rangle$.
(sync) A rule to synchronize on a barrier. The two threads that are synchronizing exchange the values v_1 and v_2. This step removes the barrier.
(exit) A rule for removing finished threads from the configuration.
(frame) A rule for extending the preceding rules to larger configurations, by allowing the rest of the configuration to pass through unchanged.

This is a possible execution of the second example from Section 2:

$$\left\{ 0 \mapsto \mathsf{Thread} \begin{pmatrix} \textbf{let}\ x = \textbf{fork}(\lambda x'.\ \textbf{print}(x'\ 1)) \\ \textbf{let}\ y = \textbf{fork}(\lambda y'.\ y'\ x) \\ \textbf{print}(1 + y\ ()\ 0) \end{pmatrix} \right\} \overset{0}{\rightsquigarrow}$$

$$\begin{Bmatrix} 0 \mapsto \mathsf{Thread} \begin{pmatrix} \textbf{let}\ y = \textbf{fork}(\lambda y'.\ y'\ \ \langle 1\rangle) \\ \textbf{print}(1 + y\ ()\ 0) \end{pmatrix} \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \end{Bmatrix} \overset{0}{\rightsquigarrow} \begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + \langle 3\rangle\ \ ()\ 0)) \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \\ 3 \mapsto \mathsf{Barrier} \\ 4 \mapsto \mathsf{Thread}\,(\langle 3\rangle\ \ \langle 1\rangle) \end{Bmatrix} \overset{3}{\rightsquigarrow}$$

$$\begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + \langle 1\rangle\ \ 0)) \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \\ 4 \mapsto \mathsf{Thread}\,(()) \end{Bmatrix} \overset{4}{\rightsquigarrow} \begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + \langle 1\rangle\ \ 0)) \\ 1 \mapsto \mathsf{Barrier} \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(\langle 1\rangle\ \ 1)) \end{Bmatrix} \overset{1}{\rightsquigarrow}$$

$$\begin{Bmatrix} 0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + 1)) \\ 2 \mapsto \mathsf{Thread}\,(\textbf{print}(0)) \end{Bmatrix} \overset{2*}{\rightsquigarrow} \{0 \mapsto \mathsf{Thread}\,(\textbf{print}(1 + 1))\} \overset{1*}{\rightsquigarrow} \{\}$$

For simplicity, we treat **print** on natural numbers as a no-op that returns (), instead of adding an output log to the semantics, because whether or not **print** logs its output somewhere does not affect the further execution of the program.

While untyped ƛ programs can easily get stuck, for instance if one side throws away its barrier, or sets up cyclic waiting dependencies, well-typed ƛ programs never get stuck. More formally, we prove *global progress* (in Section 5), which means that if we start with an initial program $e : \mathbf{1}$, then any non-empty configuration we can reach from e can step further. But first, we will see how to encode session types in ƛ.

4 Encoding session types in ƛ

Despite being very simple, ƛ's type system can encode session types. There are five basic session type constructors:

$$s \in \mathit{Session} \triangleq \mathsf{End} \mid \mathbf{!}\tau.s \mid \mathbf{?}\tau.s \mid s \oplus s \mid s \,\&\, s$$

The type $\mathbf{!}\tau.s$ means to send a value of type τ and then continue with s. Dually, $\mathbf{?}\tau.s$ means to receive a value of type τ and then continue with s. The type $s_1 \oplus s_2$ indicates that we have a choice of continuing either with protocol s_1 or with s_2. Dually, the type $s_1 \,\&\, s_2$ means that we receive a choice from the other side: either we have to continue with protocol s_1 or with protocol s_2, depending on what the other side chose. Lastly, the protocol End means that we are done with the channel and we must deallocate it. Session types make the notion of duality explicit using the function $\mathsf{dual} : \mathit{Session} \to \mathit{Session}$:

$$\begin{aligned}
\mathsf{dual}(\mathsf{End}) &\triangleq \mathsf{End}\\
\mathsf{dual}(\mathbf{!}\tau.s) &\triangleq \mathbf{?}\tau.\mathsf{dual}(s)\\
\mathsf{dual}(\mathbf{?}\tau.s) &\triangleq \mathbf{!}\tau.\mathsf{dual}(s)\\
\mathsf{dual}(s_1 \oplus s_2) &\triangleq \mathsf{dual}(s_1) \,\&\, \mathsf{dual}(s_2)\\
\mathsf{dual}(s_1 \,\&\, s_2) &\triangleq \mathsf{dual}(s_1) \oplus \mathsf{dual}(s_2)
\end{aligned}$$

The idea is that if our channel has type s, then the channel of the party we are communicating with has type $\mathsf{dual}(s)$. This is the list of channel operations and their types:

fork$_{GV}$: $(s \multimap 1) \to \mathsf{dual}(s)$
Fork off a new thread running the closure. Passes a channel of type s to the child thread, and returns a channel of type $\mathsf{dual}(s)$ to the main thread.

close : End $\to$ 1
Close and deallocate the channel. Returns unit ().

send : $\mathbf{!}\tau.s \times \tau \to s$
Send a message of type τ to the channel. Returns a new channel of type s for performing the rest of the protocol.

receive : $\mathbf{?}\tau.s \to s \times \tau$
Receive a message from the channel. Returns a pair $s \times \tau$ of the channel for performing the rest of the protocol (type s) and the message received (type τ).

tell$_\mathrm{L}$: $s_1 \oplus s_2 \to s_1$
In a branching protocol, choose the left branch. Returns a channel of the chosen type.

tell$_\mathrm{R}$: $s_1 \oplus s_2 \to s_2$
In a branching protocol, choose the right branch. Returns a channel of the chosen type.

ask : $s_1 \,\&\, s_2 \to s_1 + s_2$
Receives the choice made by the other side. Returns a sum type, which is $\mathbf{in_L}(c)$ with $c : s_1$ if the left branch was chosen by the other side, and $\mathbf{in_R}(c)$ with $c : s_2$ if the right branch was chosen.[5]

We will encode channels as ƛ's barriers, and we therefore encode a session type s as a linear function type $\tau_1 \multimap \tau_2$ where τ_1, τ_2 are determined from s. Intuitively, the sending side not only transfers the values specified by the protocol, but also a continuation channel for the remainder of the protocol. The continuation channel is connected to a tiny messenger thread, which is responsible for synchronizing with the old barrier, as we did in Section 2. We define an encoding function $[\![\cdot]\!] : \mathit{Session} \to \mathit{Type}$ that converts a session type to a ƛ type. The encoding of session types into ƛ types is as follows:

$$
\begin{aligned}
[\![\mathsf{End}]\!] &\triangleq \mathbf{1} \multimap \mathbf{1} \\
[\![!\tau.s]\!] &\triangleq [\![\mathsf{dual}(s)]\!] \times \tau \multimap \mathbf{1} \\
[\![?\tau.s]\!] &\triangleq \mathbf{1} \multimap [\![s]\!] \times \tau \\
[\![s_1 \oplus s_2]\!] &\triangleq [\![\mathsf{dual}(s_1)]\!] + [\![\mathsf{dual}(s_2)]\!] \multimap \mathbf{1} \\
[\![s_1 \,\&\, s_2]\!] &\triangleq \mathbf{1} \multimap [\![s_1]\!] + [\![s_2]\!]
\end{aligned}
$$

Using this encoding, we can implement channel operations with type signatures matching their native session-typed version, provided we use the encoding:

$$
\begin{aligned}
\mathbf{fork}_{GV} &: ([\![s]\!] \multimap \mathbf{1}) \to [\![\mathsf{dual}(s)]\!] & \mathbf{fork}_{GV}(x) &\triangleq \mathbf{fork}(x) \\
\mathbf{close} &: [\![\mathsf{End}]\!] \to \mathbf{1} & \mathbf{close}(c) &\triangleq c\ () \\
\mathbf{send} &: [\![!\tau.s]\!] \times \tau \to [\![s]\!] & \mathbf{send}(c, x) &\triangleq \mathbf{fork}(\lambda c'.\ c\ (c', x)) \\
\mathbf{receive} &: [\![?\tau.s]\!] \to [\![s]\!] \times \tau & \mathbf{receive}(c) &\triangleq c\ () \\
\mathbf{tell_L} &: [\![s_1 \oplus s_2]\!] \to [\![s_1]\!] & \mathbf{tell_L}(c) &\triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_L}(c')) \\
\mathbf{tell_R} &: [\![s_1 \oplus s_2]\!] \to [\![s_2]\!] & \mathbf{tell_R}(c) &\triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_R}(c')) \\
\mathbf{ask} &: [\![s_1 \,\&\, s_2]\!] \to [\![s_1]\!] + [\![s_2]\!] & \mathbf{ask}(c) &\triangleq c\ ()
\end{aligned}
$$

The fork operation for channels simply delegates to the fork operation of ƛ, because a channel is represented as a barrier.

Formal statement of well-typedness of the encodings

You may note that while there is an encoding function $[\![\cdot]\!]$ of session types into ƛ types, there is no explicit encoding function of GV terms to ƛ terms. This is intentional: because the translation is *local*, the definitions above can be viewed as *syntactic abbreviations* or *macros*. For instance, we can define the abbreviation

$$\mathbf{tell_L} \triangleq \lambda c.\ \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_L}(c'))$$

[5] In the original GV, branching was combined with receiving a choice. We decouple them, and let receiving a choice return a sum type, which can subsequently be pattern matched on using **match**.

of the $\mathbf{tell_L}$ channel operation as a closed syntactic ƛ term. We can then *prove* that for all session types s_1 and s_2, the typing judgement

$$\emptyset \vdash \mathbf{tell_L} : [\![s_1 \oplus s_2]\!] \to [\![s_1]\!]$$

for the $\mathbf{tell_L}$ term given above is derivable from ƛ's typing rules. The most interesting case is **fork**; in order to prove

$$\emptyset \vdash \mathbf{fork} : ([\![s]\!] \multimap \mathbf{1}) \to [\![\mathsf{dual}(s)]\!]$$

for all session types s, we rely on the following lemma about dual and the encoding $[\![\cdot]\!]$:

$$([\![s]\!] = \tau_1 \multimap \tau_2) \iff ([\![\mathsf{dual}(s)]\!] = \tau_2 \multimap \tau_1)$$

That is, if the session types are dual in the session types sense, then their encodings are dual in the ƛ sense.

The advantage of this approach is its simplicity and that we can freely intermix channels with direct usage of barriers in the same program. However, for our simulation result (Section 4.1), we do need an explicit definition of GV syntax and a translation of GV terms to ƛ terms.

A note on the mechanization and n-ary choice

We can combine n-ary choice with sending/receiving a message in a single communication step using n-ary sum types:

$$\begin{aligned} \mathbf{sendchoice_i} &: \mathbf{!}\{i : \tau_i.s_i\}_{i\in I} \times \tau_i \to s_i \\ \mathbf{receivechoice} &: \mathbf{?}\{i : \tau_i.s_i\}_{i\in I} \to \Sigma_{i\in I}\, s_i \times \tau_i \end{aligned}$$

This can also be encoded in ƛ, and is what is provided by the mechanization (7):

$$\begin{aligned} \mathbf{sendchoice_i}(c, x) &\triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_i}(c', x)) \\ \mathbf{receivechoice}(c) &\triangleq c\ () \end{aligned}$$

Encoding ƛ in GV

We can also do the encoding the other way around, and implement ƛ's fork in terms of GV's channel constructs:

$$\begin{aligned} &\mathbf{fork}_{ƛ}(f) \triangleq \\ &\quad \mathbf{let}\ c_1 = \mathbf{fork}_{GV}(\lambda c_1'.\ f\ (\lambda v'.\ \mathbf{let}\ c_2' = \mathbf{send}(c_1', v') \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ (c_3', v) = \mathbf{receive}(c_2') \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{close}(c_3');\ v)) \\ &\quad \lambda v.\ \mathbf{let}\ (c_2, v') = \mathbf{receive}(c_1) \\ &\qquad\quad \mathbf{let}\ c_3 = \mathbf{send}(c_2, v) \\ &\qquad\quad \mathbf{close}(c_3);\ v' \end{aligned}$$

Given how short the encodings of GV's channel operations in ƛ are, it is perhaps surprising that the other way around requires comparatively more code.

4.1 Simulation of GV's semantics with ƛ's semantics

To show that the encoding makes sense, we prove that we can simulate an asynchronous version of GV using ƛ's semantics. We use an asynchronous semantics ($\rightsquigarrow_{GV}$) for GV, so the GV configuration contains buffers. For details about the GV semantics used in the proof, we refer the reader to the mechanization (Section 7). The key idea behind the simulation is that each message in a buffer on the GV side corresponds to a messenger thread on the ƛ side. Whenever a message is put in a buffer on the GV side, a messenger thread is created on the ƛ side, and the messenger thread will be waiting to synchronize with a barrier. Whenever a message is received from a channel's buffer on the GV side, the receiver and the messenger thread execute their sync operation on the ƛ side, which sends the message to the receiver and allows the messenger thread to terminate.

Formally, we start with an encoding $[\![e]\!]$ that translates GV terms to the corresponding ƛ terms by replacing all occurrences of GV channel operations with their ƛ definitions given above. We then extend this translation to configurations $[\![\rho]\!]$, The translation on configurations replaces each buffer in the GV heap with a sequence of ƛ messenger threads, with one messenger thread per message in the buffer. With these notions at hand, we can show that the ƛ encodings simulate the GV semantics.

▶ **Lemma 1** (Simulation). *If* $\rho \rightsquigarrow_{GV} \rho'$ *then* $[\![\rho]\!] \rightsquigarrow^* [\![\rho']\!]$

This lemma has been mechanized in Coq (Section 7). We need the transitive closure ($\rightsquigarrow^*$) on the ƛ side, because a single step in the GV semantics can correspond to multiple steps in the ƛ semantics, since the ƛ semantics does extra administrative β-reductions. For instance, when the GV program does a $\textbf{send}(c, v)$ operation, it places the message v in the buffer in one step. The translated ƛ program on the other hand spawns a messenger thread with $[\![\textbf{send}(c, v)]\!] = \textbf{fork}(\lambda c'.\ c\ (c', v))$, which initializes the new thread with the term $(\lambda c'.\ c\ (c', v))\,\langle k\rangle$ where $\langle k\rangle$ is the newly created barrier. The messenger thread then performs the β-reduction, resulting in an extra step on the ƛ side.

To get a full operational correspondence [14], we need a second "reflection" lemma stating that if the image of the translation $[\![\rho]\!]$ takes a step, then this step can be matched with a corresponding step in the GV semantics. Note that this only holds for well-typed terms: if we have the ill-typed term $\textbf{receive}(\lambda x.\ x)$ in the GV source, this is translated to well-typed $(\lambda x.\ x)()$ in ƛ. Thus, whereas $\textbf{receive}(\lambda x.\ x)$ gets stuck in the GV semantics, its translation does not get stuck in the ƛ semantics. We do expect a full operational correspondence to hold for well-typed terms, but while we have mechanized the proof of the simulation direction (Lemma 1), we have not mechanized a full operational correspondence. With a full operational correspondence it would be possible to lift ƛ's deadlock freedom result to GV, and it would be interesting to see whether using ƛ as a "proof IR" in this manner is a viable strategy for proving deadlock freedom of GV.

4.2 Summary

To add GV's session types to linear λ-calculus, we need to add the 5 new session type formers, the notion of duality, and the 7 session type operations. In contrast, ƛ only adds one new operation, **fork**, and no new type formers and no notion of duality. Nevertheless, we have seen that we can encode session types in ƛ.

The encoding creates a new thread to store each message. Thus, ƛ should not be viewed as a practical way to implement session types, but as a theoretical calculus, like other calculi that create one thread per message, such as the asynchronous π-calculus.

5 Deadlock freedom, leak freedom, and global progress

Linear typing in ƛ guarantees strong properties for well-typed programs:

Type safety: threads never get stuck, except by synchronizing with a barrier.
Global progress: a non-empty configuration can always take a step.
Strong deadlock freedom: no subset of the threads gets stuck by waiting for each other.
Memory leak freedom: all barriers in the configuration remain referenced by a thread.

These properties are all inequivalent in strength: none of the 4 properties is strictly stronger than another. In Section 5.3 we consider a property that is strictly stronger than these 4, but we will first focus on *global progress* (Section 5.1), as ideas behind the proof of global progress (Section 5.2), are also sufficient to prove the stronger property (Section 5.3).

5.1 Global progress

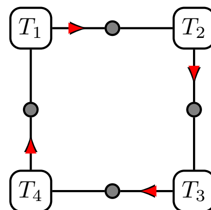
Let us consider the formal statement of global progress:

▶ **Theorem 2** (Global progress).
If $\emptyset \vdash e : \mathbf{1}$*, and* $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho$*, then either* $\rho = \{\}$ *or* $\exists \rho'.\, \rho \rightsquigarrow \rho'$*.*

Intuitively, global progress states that if we start with a well-typed program, then any configuration we reach is either empty (*i.e.,* all threads have terminated and all barriers have been deallocated), or the configuration can perform a step. This property relies on linear typing, as ƛ programs that violate linearity can deadlock and create a non-empty configuration that cannot step. A simple example is if one side does not use its barrier:

$\mathbf{let}\ x = \mathbf{fork}(\lambda x'.\ ())\ \mathbf{in}\ x\ 0$ Deadlock!

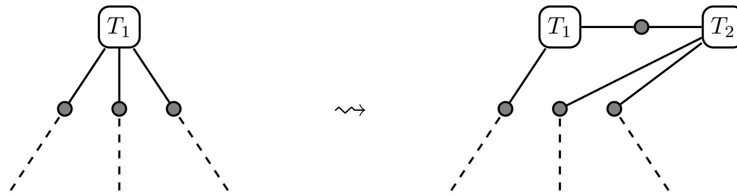
This deadlock is prevented by the linear type system, which ensures that each barrier is used *exactly once*. More complicated deadlocks are also possible in untyped programs, in which there is a circle of threads T_1 T_2 T_3 T_4 connected by barriers ● ● ● ● that are trying to synchronize (→) in a cycle:



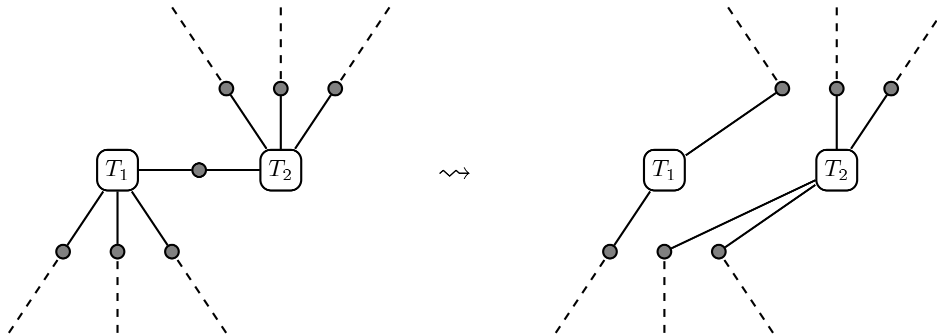
No thread can make progress because the threads are all synchronizing on different barriers. This shows that a simplistic scheme to prove deadlock freedom cannot work: we must somehow rule out such cycles. Fortunately, the linear type system ensures that the graph of connections between threads has the shape of a *forest* (*i.e.,* collection of trees, *i.e.,* an acyclic graph), and thus such circular deadlocks cannot happen. To see why the graph remains acyclic, consider what happens when we **fork**:

$\mathbf{let}\ x_1 = \mathbf{fork}(...)$
$\mathbf{let}\ x_2 = \mathbf{fork}(...)$
$\mathbf{let}\ x_3 = \mathbf{fork}(...)$
$\mathbf{let}\ y = \mathbf{fork}(\lambda y'.\ \cdots x_2 \cdots x_3 \cdots)$
$\cdots x_1 \cdots$

At the fourth **fork**, the barriers x_2 and x_3 are transferred to the new thread via lexical scoping, while the main thread keeps x_1 for itself. This is what happens to the graph:



On the left hand side, we have the thread T_1 that is about to perform the fourth **fork**. It currently owns 3 barriers x_1, x_2, x_3, which are connected to the rest of the graph. After the fork, we have the new thread T_2, which is connected to T_1 by means of a new barrier. Crucially, the barriers x_2 and x_3 of the barriers that T_1 used to own were transferred to T_2 by means of lexical scoping. Nevertheless, as one can see in the figure above, if the graph of the configuration before the fork was acyclic, then the graph of the configuration after the fork is also acyclic. The same applies to a synchronization step, when values containing potentially multiple barriers are exchanged:



On the left, T_1 and T_2 each own 3 barriers, and they are also connected by a barrier. On the right, after the synchronization has taken place, the barrier between them has disappeared. Two of the barriers of T_1 were transferred to T_2, and one of the barriers of T_2 was transferred to T_1. Once again, if the graph on the left was acyclic, then the graph on the right will still be acyclic.[6]

The other operations of λ do not change the connections in the graph. Therefore, a program starts with a single thread, and then grows and alters its graph in a dance of fork and sync steps, but the graph remains acyclic at all times.

In the next section we will see in a bit more detail how the acyclicity of the graph is used in the proof of global progress.

Related work Graph theoretic deadlock-freedom arguments are common in the session types literature and have previously been made by Carbone [5], Lindley and Morris [21], and Fowler, Kokke, Dardha, Lindley, and Morris [10].

---

[6] When one looks at the picture of the synchronizing threads, it seems that T_1 becomes totally disconnected from T_2 after the synchronization. This means that the threads can only communicate *once*. Yet the session-typed channel encoding seems to make it possible to communicate several times. Exercise for the reader: what is the solution to this paradox?

$$\frac{\cdot}{k:\tau;\emptyset \vdash \langle k \rangle : \tau} \qquad \frac{\cdot}{\emptyset; x:\tau \vdash x : \tau} \qquad \frac{\Sigma; \Gamma, x:\tau_1 \vdash e : \tau_2}{\Sigma; \Gamma \vdash \lambda x.\ e : \tau_1 \multimap \tau_2}$$

$$\frac{\Sigma_1; \Gamma_1 \vdash e_1 : \tau_1 \multimap \tau_2 \qquad \Sigma_2; \Gamma_2 \vdash e_2 : \tau_1}{\Sigma_1, \Sigma_2; \Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

Figure 3 Run-time type system (selected rules).

5.2 Structure of the global progress proof

This section gives more detail about how the acyclicity of the connection graph is used to prove global progress. For the full details, the interested reader is referred to the mechanization (Section 7).

Global progress states that if the configuration is non-empty, then it can take a step. Formally, there are several types of stuck configurations to rule out. Perhaps the configuration has a single thread and no barriers, but the thread is stuck on a type error (violating type safety). Or perhaps the configuration consists of just one barrier and no threads (violating memory leak freedom). Or perhaps there is a single thread and a single barrier, and the thread is trying to synchronize but is stuck due to the absence of a partner to synchronize with (violating deadlock freedom).

The aforementioned stuck configurations and more complicated variations are ruled out by keeping track of sufficient type information and local invariants for each object in the configuration (*e.g.,* that a barrier is always connected to exactly two threads, and that the types of the references to the barrier are dual $\tau_1 \multimap \tau_2$ and $\tau_2 \multimap \tau_1$). The interesting case is when the types are all locally correct, but the configuration is still deadlocked due to cyclic waiting. This case is ruled out by proving that well-typed programs maintain the invariant that the connection graph is acyclic, which implies that cyclic waiting cannot happen.

Formally, we define a *well-formedness* invariant of configurations, that maintains well-typedness of the threads as well as the acyclicity of the connection graph. For the well-typedness, we use the run-time type system in Figure 3.[7] The rules of the run-time type system correspond to the rules of the static type system, plus one additional rule for typing barrier literals $\langle k \rangle$. The typing judgment uses an additional Σ-context for typing those barrier literals.

We say that a configuration ρ is *well-formed* if we have an *acyclic* graph G (*i.e.,* an undirected forest) where the vertices correspond to the entries of the configuration, and the edges (between threads and barriers) are labeled with types, such that for each vertex i in G:

- If $\rho(i) = \mathsf{Thread}(e)$ then $\Sigma; \emptyset \vdash e : \mathbf{1}$, where Σ is given by the types on the edges of the barriers connected to vertex i.
- If $\rho(i) = \mathsf{Barrier}$ then vertex i has edges to two different threads, labeled with dual types $\tau_1 \multimap \tau_2$ and $\tau_2 \multimap \tau_1$.

These conditions ensure that the graph structure matches the structure of the configuration: if we have a barrier literal $\langle k \rangle$ somewhere in the expression e of thread n, then the first condition ensures that we have an edge between n and k, labeled with a type $\tau_1 \multimap \tau_2$

[7] Our run-time type system in Coq makes use of separation logic, following [18]. This is equivalent to the type system in the figure, but easier to work with in a proof assistant.

to make expression e well-typed. The second condition then ensures that there is a second thread with barrier literal $\langle k \rangle$ in its expression, with type $\tau_2 \multimap \tau_1$. Note that the two occurrences of the very same barrier literal $\langle k \rangle$ have two different types in the two different threads.

Now that the configuration invariant has been defined, proof of global progress (Theorem 2) can be structured as follows. Using well-typedness, we are able to show that the only way a configuration can get stuck is if all threads are trying to synchronize with a barrier. The invariant maintains that the graph structure is always acyclic. By a mathematical argument about graphs and the pigeonhole principle, we are then able to show that two of the threads must be synchronizing on the same barrier. Hence, at least one synchronization step can proceed, and we have global progress.

5.3 Strengthened deadlock and memory leak freedom

Global progress (Theorem 2) rules out whole-program deadlocks. It also ensures that all barriers have been used when the program finishes. However, this theorem does not guarantee anything as long as there is still a single thread that can step. Thus it does not guarantee local deadlock freedom, nor memory leak freedom while the program is still running, and it does not even guarantee type safety: a situation in which a thread is stuck on a type error is formally not ruled out by this theorem as long as there is another thread that can still step.

Our goal is to find a formulation that is strictly stronger than these 4 properties, and from which they can be easily proved as corollaries. We take inspiration from [18], and find a strengthened formulation of deadlock freedom on the one hand, and strengthened memory leak freedom on the other hand. These strengthened formulations of deadlock freedom and memory leak freedom are equivalent to each other, and they imply type safety and global progress. In order to state these, we need the relation $i \ \mathsf{waiting}_\rho \ j$, which says that $i \in \mathsf{dom}(\rho)$ is waiting for $j \in \mathsf{dom}(\rho)$.

Intuitively, the meaning of $\mathsf{waiting}_\rho$ is as follows. When a barrier k gets allocated, the literal $\langle k \rangle$ appears in two threads n_1, n_2. In this case we say that $k \ \mathsf{waiting}_\rho \ n_1$ and $k \ \mathsf{waiting}_\rho \ n_2$, because the barrier is waiting until the threads want to synchronize with it. Note that this relationship is dynamic: if the barrier literal $\langle k \rangle$ is transferred to another thread, then the barrier is waiting for that new thread to synchronize with it. Whenever a thread n starts trying to synchronize with k by calling $\langle k \rangle \ v$ for some value v, then the waiting relationship flips: we now say that the thread is waiting for the barrier, *i.e.,* that $n \ \mathsf{waiting}_\rho \ k$. Formally:

▶ **Definition 3.** *We have* $i \ \mathsf{waiting}_\rho \ j$ *if either:*
1. $\rho(i) = \mathsf{Barrier}$ *and* $\rho(j) = \mathsf{Thread}(e)$ *and* $\langle i \rangle \in e$, *but* $e \neq K[\ \langle i \rangle \ v\ ]$, *or*
2. $\rho(i) = \mathsf{Thread}(e)$ *and* $\rho(j) = \mathsf{Barrier}$, *and* $e = K[\ \langle j \rangle \ v\ ]$

Using this notion, we can define what a partial deadlock/leak is. Intuitively, a partial deadlock is a situation in which there is some subset of the threads that are all waiting for each other. Because our notion of waiting also incorporates barriers, we generalize this to say that a *partial deadlock/leak* is a situation in which there is some subset of the threads and barriers that are all waiting for each other. Formally:

▶ **Definition 4** (Partial deadlock/leak). *Given a configuration* ρ*, a non-empty subset* $S \subseteq \mathsf{dom}(\rho)$ *is in a partial deadlock/leak if these two conditions hold:*
1. *No* $i \in S$ *can step, i.e., for all* $i \in S$, $\neg \exists \rho'.\, \rho \overset{i}{\rightsquigarrow} \rho'$
2. *If* $i \in S$ *and* $i \ \mathsf{waiting}_\rho \ j$ *then* $j \in S$

This notion also incorporates memory leaks: if there is some barrier that is not referenced by a thread, then the singleton set of that barrier is a partial deadlock/leak. Similarly, a single thread that is not synchronizing on a barrier, is considered to be in a singleton deadlock if it cannot step. This way, the notion of partial deadlock incorporates type safety.

▶ **Definition 5** (Partial deadlock/leak freedom). *A configuration ρ is deadlock/leak free if no $S \subseteq \mathsf{dom}(\rho)$ is in a partial deadlock/leak.*

We also strengthen the standard notion of memory leak freedom, namely reachability, to incorporate aspects of deadlock freedom.

▶ **Definition 6** (Reachability). *We inductively define the threads and barriers that are* reachable *in ρ: $j_0 \in N$ is reachable in ρ if there is some sequence $j_1, j_2, ..., j_k$ (with $k \geq 0$) such that j_0 $\mathsf{waiting}_\rho$ j_1, and j_1 $\mathsf{waiting}_\rho$ j_2, ..., and j_{k-1} $\mathsf{waiting}_\rho$ j_k, and finally j_k can step in ρ, i.e., $\exists \rho'.\, \rho \overset{j_k}{\rightsquigarrow} \rho'$.*

Intuitively, an element $j_0 \in \mathbb{N}$ is reachable if j_0 can itself step or has a transitive waiting dependency on some j_k that can step. This notion is stronger than the usual notion of reachability, which considers objects to be reachable even if they are only reachable from threads that are blocked.

▶ **Definition 7** (Full reachability). *A configuration ρ is* fully reachable *if all $i \in \mathsf{dom}(\rho)$ are reachable in ρ.*

As in [18], our strengthened formulations of deadlock freedom and full reachability are equivalent for $\bar{\lambda}$:

▶ **Theorem 8.** *A configuration ρ is deadlock/leak free if and only if it is fully reachable.*

Furthermore, these notions imply global progress and type safety:

▶ **Definition 9.** *A configuration ρ has the progress property if $\rho = \emptyset$ or $\exists \rho', i.\, \rho \overset{i}{\rightsquigarrow} \rho'$.*

▶ **Definition 10.** *A configuration ρ is safe if for all $i \in \mathsf{dom}(\rho)$, either $\exists \rho', i.\, \rho \overset{i}{\rightsquigarrow} \rho'$, or $\exists j.\, i$ $\mathsf{waiting}_\rho$ j.*

▶ **Theorem 11.** *If a configuration ρ is deadlock/leak free (or equivalently, fully reachable), then ρ has the progress and safety properties.*

Our main theorem is thus that configurations that arise from well-typed programs are fully reachable and deadlock free:

▶ **Theorem 12.** *If $\emptyset \vdash e : \mathbf{1}$ and $\{0 \mapsto \mathsf{Thread}(e)\} \rightsquigarrow^* \rho'$, then ρ' is fully reachable and deadlock/leak free.*

The proof of the reachability half of Theorem 12 proceeds similarly to the proof of global progress (Theorem 2). The difference between the proofs is that the proof of reachability needs to explicitly keep track of the reason why each object in the configuration is reachable, whereas global progress only needs to find one of the “roots” of the reachability relation (*i.e.*, threads that can step).

Theorem 8 can then be used to obtain the deadlock freedom side of Theorem 12. The idea of the proof of Theorem 8 is that the set of all unreachable objects forms a deadlock, if the set is non-empty.

6 Extending λ with unrestricted and recursive types

We add unrestricted types and recursive types as extensions. These can be omitted for a minimalistic language, but they enable us to do ordinary functional programming and recursive sessions in λ, bringing it closer to a realistic language in terms of features and expressiveness. The extended set of types is:

$$\tau \in \mathit{Type} ::= \mathbf{0} \mid \mathbf{1} \mid \tau \times \tau \mid \tau + \tau \mid \tau \multimap \tau \mid \tau \to \tau \mid \mu a.\tau \mid a$$

An equi-recursive interpretation of $\mu x.\tau$ avoids explicit (un)fold constructs [6]. Recursive types make the language Turing complete, because they allow us to define recursive functions with the Y-combinator[8], and together with sums and products can be used to encode algebraic data types. Because session types are encoded as ordinary types in λ, recursive sessions are automatically supported. This includes recursion through the message, as in $\mu s.\ ?s.\mathsf{End}$, which is encoded as $\mu a.\ \mathbf{1} \multimap (\mathbf{1} \multimap \mathbf{1}) \times a$.

Formally and in the mechanization (Section 7), we use the coinductive method of Gay, Thiemann, and Vasconcelos [12] to handle equi-recursive types. This means that we formally do not have a syntactic $\mu a.\tau$ type constructor; instead we let the language of types be *coinductively generated*. Intuitively, this means that infinite types are allowed, and a recursive type $\mu a.F(a)$ is represented as its infinite unfolding $F(F(F(\cdots)))$. We use a meta-level fixpoint (CoFixpoint in Coq) to construct infinite/circular types. By using this method we do not need an additional typing rule for unfolding recursive types, since types are already identified up to unfolding.

In order to make interesting use of recursive types, it is necessary to add *unrestricted types*, which are types for which the linearity restriction is lifted, so that they can be duplicated and discarded freely. A linear function can only be called once and hence cannot call itself, so recursive functions must have unrestricted type. Using unrestricted and recursive types, we do not need built-in recursion as we can encode recursive functions using the Y-combinator:

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

Here too it is apparent that since x is used twice, unrestricted as well as recursive types are required to type check the Y-combinator [18].

In particular, we add the unrestricted function type $\tau_1 \to \tau_2$ as a new type former. We can then define rules that determine which of the existing types are unrestricted, as follows:

- $\mathbf{0}, \mathbf{1}$ are unrestricted types
- $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$ are unrestricted if τ_1 and τ_2 are unrestricted
- $\tau_1 \to \tau_2$ is always unrestricted, even for linear τ_1 and τ_2
- $\tau_1 \multimap \tau_2$ is always linear, even for unrestricted τ_1 and τ_2

Using unrestricted function types, we can encode the $!A$ connective from linear logic as $\mathbf{1} \to A$, and $?A$ as $A \to \mathbf{1}$.

Formally and in the mechanization, unrestricted types are handled by splitting the typing context using a 3-part relation $\mathsf{split}\ \Gamma\ \Gamma_1\ \Gamma_2$, which intuitively says that $\Gamma \equiv \Gamma_1, \Gamma_2$, where variables of unrestricted type in Γ may occur in both Γ_1 and Γ_2 (see [18] for details). We also make use of the predicate $\Gamma\ \mathsf{unr}$ to indicate that all variables in Γ must have unrestricted type. To extend the typing rules in Figure 1, we use split to split the context, and we use

8 One could also add an explicit **letrec**.

Γ unr wherever an empty context is required in Figure 1. For example, here are the typing rules for variables and for pairs:

$$\frac{\Gamma\ \mathsf{unr}}{\Gamma, x{:}\tau \vdash x : \tau} \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2 \qquad \mathsf{split}\ \Gamma\ \Gamma_1\ \Gamma_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

The other rules and the rules of the run-time type system are amended analogously.

Unrestricted and recursive types are purely type-level features and require no extensions to the expression language or to the operational semantics. Nevertheless, they upgrade $\barwedge$ to a Turing complete functional and concurrent language. For more details, we refer the interested reader to the mechanization (Section 7).

7 Mechanization

All our theorems have been mechanized in Coq. We use the connectivity graph library of [18] in our mechanization. The mechanization is built-up as follows:

- Language definition: expressions, static type system, and operational semantics. Our mechanization includes the extensions with unrestricted and recursive types.
- A run-time type system that extends the static type system to barrier literals $\langle k \rangle$. The run-time type system is expressed in separation logic.
- A configuration well-formedness invariant, stating that the configuration remains well-typed, and the connectivity between threads and barriers remains acyclic.
- Proof that well-formedness is maintained by the operational semantics.
- Proof that well-formed configurations have the *full-reachability* property.
- Proofs that full-reachability is equivalent to deadlock/leak freedom, and that they imply type safety and global progress.
- The definition of the encoding of session types in $\barwedge$, and proofs that the usual session typing rules are admissible.
- A definition of GV and its operational semantics, and the translation into $\barwedge$, with a proof that the GV semantics can be simulated with the $\barwedge$ semantics. A lock-step simulation is obtained by inserting extra no-op steps in the GV semantics wherever $\barwedge$ does an administrative β-reduction.

Whereas the mechanized deadlock freedom proof for GV's session types by [18] consists of 2139 lines of Coq definitions and proofs (excluding [18]'s graph library), our mechanization of $\barwedge$ and its deadlock freedom is only 1229 lines. The encoding of GV's session types into $\barwedge$ together with the proofs of admissibility of the typing rules is 249 lines, and the operational simulation result is 309 lines.

Although the $\barwedge$ mechanization relies on connectivity graphs [18], the techniques presented there were not immediately sufficient for proving deadlock freedom of $\barwedge$. The difficulty lies in $\barwedge$'s sync step in the operational semantics, which exchanges resources between two vertices that are not directly adjacent in the graph. This is not supported as an operation by [18], so we instead want to separate it into multiple smaller graph transformations. Unfortunately, the intermediate states do not satisfy the configuration invariant. The solution to this was to add extra "ghost state" to the labels on the edges of the graph, which keeps track of which sub-step of the decomposed graph transformation the connected vertices are in. As part of future work, it would be interesting to investigate whether this technique can be used more generally for composing graph transformations on the separation logic level, when the intermediate states do not satisfy the invariant.

The current version of the mechanization can be found on GitHub [17].

8 Related work

Session types were originally described by Honda [15], and later by Honda, Vasconcelos, and Kubo [16]. Gay and Vasconcelos [13] embedded session types in a linear λ-calculus. Whereas Gay and Vasconcelos' calculus [13] was not yet deadlock free, Wadler's subsequent GV [31] and its derivatives [21, 23, 24, 11, 10] were.

Wadler also described the relation of GV to classical processes (CP) [31], giving a kind of Curry-Howard correspondence between session types and classical linear logic. For intuitionistic linear logic, such a correspondence had earlier been described by Caires and Pfenning [4].

Lindley and Morris [22, 23] give a CPS encoding of GV. The target of the CPS encoding is linear λ-calculus *without* fork, which is even more minimal than ƛ. The key difference between the CPS encoding and our encoding into ƛ is that the CPS encoding is *global* (*i.e.,* a whole-program transformation), whereas the encoding of sessions into ƛ is *local*, which is made possible by the built-in concurrency of ƛ. In other words, in contrast to the CPS encoding, the encoding of channel operations in ƛ can be viewed as syntactic abbreviations or macros, satisfying Felleisen's expressiveness criterion [9].

When session types are added to the syntax of standard π-calculus they give rise to additional separate syntactic categories, which leads to a duplication of effort in the theory. Kobayashi showed that session types are encodable into standard π-types [19]. Dardha, Giachino, and Sangiorgi formalize and extend Kobayashi's approach [7, 8]. The encoding makes use of the fact that the π-calculus semantics has communication in the form of π-channels, and can thus encode session communication into π-communication. The encoding of multi-step sessions into single-shot π-channels sends a continuation along, so that the communication can continue. ƛ's encoding of session types takes inspiration from this work, and also sends along continuations on which the communication can continue. On the other hand, ƛ starts with λ-calculus, which does not have any concurrency or communication. We therefore *add* concurrency and communication to linear λ-calculus in the form of fork. Unlike the π-calculus' channel communication, which is one-way (like an individual step of a session), ƛ's barrier communication atomically *exchanges* two values, so that barriers may be given linear function type $A \multimap B$. This lets ƛ get away with not adding any new type formers to linear λ-calculus, by reusing the quintessential λ-calculus type (the function type) for its communication primitive.

Single-shot synchronization primitives have also been used in the implementation of a session-typed channel libraries, for instance by Scalas and Yoshida [30], Padovani [27], and Kokke and Dardha [20].

More distantly, Arslanagic, Pérez, and Voogd developed *minimal session types* [2], which decompose multi-step session types into single-step "minimal" session types of the form $!\tau.\mathsf{End}$ and $?\tau.\mathsf{End}$ in a π-calculus. Whereas ƛ and the preceding approaches [19, 7, 8] encode sequencing by nesting payload types, minimal session types "slice" the n actions of a session s into indexed names $s_1, \ldots, s_n$, each having a minimal session type. Correct sequencing is arranged on the process level with additional synchronizations, using Parrow's decomposition of processes into trios [28].

Niehren, Schwinghammer and Smolka developed a concurrent λ-calculus with futures [26]. Futures are akin to mutable variables that can only be assigned once. If a future has not been assigned a value yet, then attempting to read its value will block until a value becomes available. In addition to a non-linear type system which allows run-time errors due to multiple assignments to the same future, the authors also present a linear type system

that ensures that futures are not assigned twice. The authors are able to build channels on top of futures by starting with the ordinary linked list data type, and making the tail of the list a future, thus making the list open-ended. Unlike session-typed channels, which follow a protocol and can send values of different types at different points in the protocol, their channels always communicate values of the same type. Besides the difference between futures (which are unidirectional) and $\lambda\!\!\!\lambda$'s barriers (which are bidirectional), another difference is that deadlock freedom is not guaranteed for all well-typed programs.

Aschieri, Ciabattoni and Genco give a Curry-Howard correspondence for Gödel Logic [3], which is intuitionistic logic extended with the axiom $(A \to B) \vee (B \to A)$[9]. This is a classical axiom that is implied by, but strictly weaker than the law of the excluded middle, and Gödel Logic thus provides an intermediate between intuitionistic and classical logic. The idea for the Curry-Howard interpretation of the axiom is that two copies of the continuation can be run in parallel, and exchange their evidence for A and B if both sides try to apply the implication obtained from the axiom. An important difference with $\lambda\!\!\!\lambda$ is that Gödel Logic is based on intuitionistic logic (corresponding to the ordinary simply typed lambda calculus), whereas $\lambda\!\!\!\lambda$ is based on the linear simply typed lambda calculus, and strongly relies on linearity for type safety and deadlock freedom. It would be interesting to investigate whether a connection between $\lambda\!\!\!\lambda$ and Gödel Logic can be established, but a naive attempt at interpreting the axiom as $(A \to B) + (B \to A)$ in $\lambda\!\!\!\lambda$ appears bound to fail, because in a linear setting the continuation/context cannot be duplicated without breaking the meta-theoretical properties.

We base our mechanization on the *connectivity graph* approach of Jacobs, Balzer, and Krebbers [18], and we use their library to reason about graphs in Coq. This is related to the graphical approach of Carbone [5], the proof method of Lindley and Morris [21], and to the abstract process structures of Fowler, Kokke, Dardha, Lindley, and Morris [10].

More distantly, $\lambda\!\!\!\lambda$ is inspired by minimal languages such as MiniJava [29], MiniML [25], the DOT calculus [1], and others.

9 Concluding remarks

We have investigated $\lambda\!\!\!\lambda$, a minimal linear lambda calculus extended with **fork** to make it concurrent. We have seen that channel operations and linear session types can be encoded in $\lambda\!\!\!\lambda$, and we have shown that the resulting semantics for the channel operations simulates the GV semantics.

The metatheory of $\lambda\!\!\!\lambda$, including strong deadlock freedom, has been mechanized in Coq. Because of $\lambda\!\!\!\lambda$'s minimality, the proofs are simpler and shorter than earlier mechanized proofs for session types. I hope you enjoyed this approach to distilling session types into a simple core, and hope that $\lambda\!\!\!\lambda$ may serve as a minimal basis upon which future work may build.

Acknowledgements I thank Robbert Krebbers, Stephanie Balzer, Jorge Pérez, Dan Frumin, Bas van den Heuvel, Anton Golov, Ike Mulder, and last but not least, the anonymous reviewers for the helpful discussions and feedback.

[9] Thanks to Dan Frumin for pointing out this connection.

References

1 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. *The Essence of Dependent Object Types*, pages 249–272. 2016. doi:10.1007/978-3-319-30936-1_14.

2 Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. Minimal Session Types (Pearl). In *ECOOP 2019*, 2019. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10815, doi:10.4230/LIPIcs.ECOOP.2019.23.

3 Federico Aschieri, Agata Ciabattoni, and Francesco A. Genco. Gödel logic: From natural deduction to parallel computation. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005076.

4 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236, 2010. doi:10.1007/978-3-642-15375-4_16.

5 Marco Carbone and Søren Debois. A graphical approach to progress for structured communication in web services. In *ICE*, volume 38 of *EPTCS*, pages 13–27, 2010. doi:10.4204/EPTCS.38.4.

6 Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI*, pages 50–63. ACM, 1999. doi:10.1145/301618.301641.

7 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, 2012. doi:10.1145/2370776.2370794.

8 Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. *Inf. Comput.*, 256:253–286, 2017. doi:10.1016/j.ic.2017.06.002.

9 Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35–75, 1991. URL: https://www.sciencedirect.com/science/article/pii/016764239190036W, doi:https://doi.org/10.1016/0167-6423(91)90036-W.

10 Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley, and J. Garrett Morris. Separating sessions smoothly. In *CONCUR*, volume 203 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CONCUR.2021.36.

11 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: Session types without tiers. *PACMPL*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.

12 Simon J. Gay, Peter Thiemann, and Vasco T. Vasconcelos. Duality of session types: The final cut. In *PLACES*, volume 314 of *EPTCS*, pages 23–33, 2020. doi:10.4204/EPTCS.314.3.

13 Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):19–50, 2010. doi:10.1017/S0956796809990268.

14 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Information and Computation*, 208(9):1031–1053, 2010. URL: https://www.sciencedirect.com/science/article/pii/S0890540110001008, doi:https://doi.org/10.1016/j.ic.2010.05.002.

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

16 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138, 1998. doi:10.1007/BFb0053567.

17 Jules Jacobs. Coq mechanization of lambda-barrier, 2021. The most recent version is at https://github.com/julesjacobs/cgraphs. doi:https://zenodo.org/record/6560443.

18 Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: a method for proving deadlock freedom based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL):1–33, 2022. doi:10.1145/3498662.

19 Naoki Kobayashi. Type systems for concurrent programs. volume 2757 of *Lecture Notes in Computer Science*, pages 439–453, 2002. doi:10.1007/978-3-540-40007-3_26.

20 Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. Haskell 2021, 2021. doi:10.1145/3471874.3472979.

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

22 Sam Lindley and J. Garrett Morris. Embedding session types in Haskell. In *Haskell Symposium*, pages 133–145, 2016. doi:10.1145/2976002.2976018.

23 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *ICFP*, pages 434–447, 2016. doi:10.1145/2951913.2951921.

24 Sam Lindley and J. Garrett Morris. Lightweight functional session types. In *Behavioural Types: from Theory to Tools*. 2017.

25 Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ml from higher-order logic. *SIGPLAN Not.*, page 115–126, 2012. doi:10.1145/2398856.2364545.

26 Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. volume 3717 of *Lecture Notes in Computer Science*, pages 248–263, 2005. doi:10.1007/11559306_14.

27 Luca Padovani. A simple library implementation of binary sessions. *J. Funct. Program.*, 27:e4, 2017. doi:10.1017/S0956796816000289.

28 Joachim Parrow. Trios in concert. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 621–637. MIT Press, 1998.

29 Eric Roberts. An overview of minijava. *SIGCSE Bull.*, 2001. doi:10.1145/366413.364525.

30 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28, 2016. doi:10.4230/LIPIcs.ECOOP.2016.21.

31 Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012. doi:10.1145/2364527.2364568.