A Simple Concurrent Lambda Calculus for Session Types

Jules Jacobs

Radboud University Nijmegen, mail@julesjacobs.com

Abstract

We introduce μGV ("micro GV"), which strives to be a minimal extension of linear λ-calculus with concurrent communication, adding only a new **fork** construct for spawning threads. The child and parent thread communicate with each other via two dual values of linear function type $\tau_1 \xrightarrow{\text{lin}} \tau_2$ and $\tau_2 \xrightarrow{\text{lin}} \tau_1$. Using only **fork**, we can implement all of GV's channel operations and session types as a library in μGV. The linear type system ensures that μGV programs are deadlock-free and satisfy global progress, which we prove in Coq.

Session types [7, 6] for communication channels can be used to verify that programs follow the protocol specified by a channel's session type. Gay and Vasconcelos [5] embed session types in a linear λ-calculus with concurrency and channels, and Wadler's subsequent GV [13] and its derivatives [10, 11, 12, 4, 3] guarantee that all well-typed programs are deadlock free.

To add session types to linear λ-calculus, one adds session type formers and their corresponding channel 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 s), $s_1 \oplus s_2$ (send choice between s_1 and s_2), $s_1 \,\&\, s_2$ (receive choice between s_1 and s_2), and End (close channel). An example is $\mathbf{!}\tau_1.(\mathbf{?}\tau_2.\mathsf{End} \oplus \mathbf{!}\tau_3.\mathsf{End})$: send a value of type τ_1 then either receive τ_2 or send τ_3. One adds **fork** for creating a thread and a pair of dual channel endpoints for communication between the parent and child thread. For this, one needs a definition of duality, with $\mathbf{!}$ dual to $\mathbf{?}$, $\oplus$ dual to $\&$, and End dual to itself.

μGV, on the other hand, has none of these. Instead, we add only a *single* construct: **fork**.

$$\textbf{fork} : ((\tau_1 \xrightarrow{\text{lin}} \tau_2) \xrightarrow{\text{lin}} \mathbf{1}) \to (\tau_2 \xrightarrow{\text{lin}} \tau_1)$$

μGV adds no new type formers, and no explicit definition of duality. Instead, we re-use the linear function type $\tau_1 \xrightarrow{\text{lin}} \tau_2$ for communication between threads. Let us look at an example:

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

This program forks off a new thread and creates *communication barriers* c and c' to communicate between the threads. The barrier c gets returned to the main thread, and c' gets passed to the child thread. A call to a barrier will block until the other side is also trying to synchronize, and will then exchange the values passed as an argument: when $c'\ 1$ is called, it will block until $c\ 0$ is also called, and vice versa. The call $c'\ 1$ will then return 0, and the call $c\ 0$ will return 1. Thus, the program will print 0 2 or 2 0, depending on which thread prints first. Using a tiny channel library, we can write message passing programs:

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

$\textbf{let } x_1 = \textbf{fork}(\lambda x_1'.\ \textbf{let } (x_2', n_1) = \textbf{receive}(x_1')$ // receive message n_1
$\qquad \textbf{let } (x_3', n_2) = \textbf{receive}(x_2')$ // receive message n_2
$\qquad \textbf{let } x_4' = \textbf{send}(x_3', n_1 + n_2); \textbf{close}(x_4'))$ // send $n_1 + n_2$ back and close

$\textbf{let } x_2 = \textbf{send}(x_1, 1)$ // send message 1

$\textbf{let } x_3 = \textbf{send}(x_2, 2)$ // send message 2

$\textbf{let } (x_4, n) = \textbf{receive}(x_3)$ // receive $n = 1 + 2$

$\textbf{print}(n);\ \textbf{close}(x_4)$

μGV expressions and types

$$e \in \mathit{Expr} ::= x \mid () \mid (e,e) \mid \mathbf{in_L}(e) \mid \mathbf{in_R}(e) \mid \lambda x.\ e \mid e\ e \mid \mathbf{fork}(e) \mid$$
$$\mathbf{let}\ (x_1,x_2) = e\ \mathbf{in}\ e \mid \mathbf{match}\ e\ \mathbf{with}\ \ldots\ \mathbf{end}$$
$$\tau \in \mathit{Type} ::= 0 \mid 1 \mid \tau \times \tau \mid \tau + \tau \mid \tau \xrightarrow{\mathsf{lin}} \tau \mid \tau \xrightarrow{\mathsf{unr}} \tau \mid \mu x.\tau \mid x$$

Session types duality

$$\overline{\mathsf{End}} \triangleq \mathsf{End}$$
$$\overline{\mathbf{!}\tau.s} \triangleq \mathbf{?}\tau.\overline{s}$$
$$\overline{\mathbf{?}\tau.s} \triangleq \mathbf{!}\tau.\overline{s}$$
$$\overline{s_1 \oplus s_2} \triangleq \overline{s_1} \,\&\, \overline{s_2}$$
$$\overline{s_1 \,\&\, s_2} \triangleq \overline{s_1} \oplus \overline{s_2}$$

Encoding session types in μGV

$$\mathsf{End} \triangleq \mathbf{1} \xrightarrow{\mathsf{lin}} \mathbf{1}$$
$$\mathbf{!}\tau.s \triangleq \overline{s} \times \tau \xrightarrow{\mathsf{lin}} \mathbf{1}$$
$$\mathbf{?}\tau.s \triangleq \mathbf{1} \xrightarrow{\mathsf{lin}} s \times \tau$$
$$s_1 \oplus s_2 \triangleq \overline{s_1} + \overline{s_2} \xrightarrow{\mathsf{lin}} \mathbf{1}$$
$$s_1 \,\&\, s_2 \triangleq \mathbf{1} \xrightarrow{\mathsf{lin}} s_1 + s_2$$

Channel operations

$$\mathbf{fork} : (s \xrightarrow{\mathsf{lin}} \mathbf{1}) \to \overline{s}$$
$$\mathbf{close} : \mathsf{End} \to \mathbf{1}$$
$$\mathbf{send} : \mathbf{!}\tau.s \times \tau \to s$$
$$\mathbf{receive} : \mathbf{?}\tau.s \to s \times \tau$$
$$\mathbf{tell_L} : s_1 \oplus s_2 \to s_1$$
$$\mathbf{tell_R} : s_1 \oplus s_2 \to s_2$$
$$\mathbf{ask} : s_1 \,\&\, s_2 \to s_1 + s_2$$

Encoding channel operations in μGV

$$\mathbf{fork}(x) \triangleq \mathbf{fork}(x)$$
$$\mathbf{close}(c) \triangleq c\ ()$$
$$\mathbf{send}(c,x) \triangleq \mathbf{fork}(\lambda c'.\ c\ (c',x))$$
$$\mathbf{receive}(c) \triangleq c\ ()$$
$$\mathbf{tell_L}(c) \triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_L}(c'))$$
$$\mathbf{tell_R}(c) \triangleq \mathbf{fork}(\lambda c'.\ c\ \mathbf{in_R}(c'))$$
$$\mathbf{ask}(c) \triangleq c\ ()$$

Figure 1: The μGV language (top), session types (left) and their encoding in μGV (right).

As in GV, our channels are used in functional style: each channel operation returns a new channel. This channel will have a new type, reflecting the step in the protocol. In fact, GV's session types (including choice) can be encoded in terms of μGV's types, so that our channel library can be given a statically session-typed interface (see Figure 1). Recursive sessions can be encoded with recursive μGV types.

Like GV, all well-typed μGV programs are automatically *deadlock free*, and therefore satisfy *global progress*. We prove this property in Coq. Because of μGV's simplicity, these proofs are simpler and shorter than previous (mechanized) proofs for deadlock freedom of session types [8], even when counting the encoding of session types into μGV (1442 lines vs 2796 lines).

There have been other efforts for simpler systems, such as an encoding of session types into π-calculus types [2], and *minimal session types* [1], 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 library for Haskell [9]).

I hope that μGV shows that linear λ-calculus also provides a good substrate for a minimalist concurrent calculus, with communication primitives that capitalize on the fact that the quintessential linear λ-calculus type, the linear function type $\tau_1 \xrightarrow{\mathsf{lin}} \tau_2$, is a *self-dual* connective.

References

[1] Alen Arslanagic, Jorge A. Pérez, and Erik Voogd. Minimal Session Types (Pearl). In *ECOOP 2019*, 2019.

[2] Ornela Dardha, Elena Giachino, and Davide Sangiorgi. Session types revisited. In *PPDP'12*, 2012.

[3] 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.

[4] 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.

[5] Simon J. Gay and Vasco Thudichum Vasconcelos. Linear type theory for asynchronous session types. *JFP*, 20(1):19–50, 2010.

[6] Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523, 1993.

[7] 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.

[8] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. Connectivity graphs: A separation logic approach for proving deadlock freedom. In *POPL*, 2022.

[9] Wen Kokke and Ornela Dardha. Deadlock-free session types in linear haskell. Haskell 2021.

[10] Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *ESOP*, volume 9032 of *LNCS*, pages 560–584, 2015.

[11] Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *ICFP*, pages 434–447, 2016.

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

[13] Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.